

IBM DB2 Universal Database



Application Development Guide

Version 6

SC09-2845-00

IBM DB2 Universal Database



Application Development Guide

Version 6

SC09-2845-00

Before using this information and the product it supports, be sure to read the general information under "Appendix H. Notices" on page 733.

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

Order publications through your IBM representative or the IBM branch office serving your locality or by calling 1-800-879-2755 in the United States or 1-800-IBM-4YOU in Canada.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1993, 1999. All rights reserved.**

US Government Users Restricted Rights – Use duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Part 1. DB2 Application Development Concepts	1	Chapter 3. Embedded SQL Overview	43
Chapter 1. Getting Started with DB2		Embedding SQL Statements in a Host	
Application Development	3	Language	43
About This Book	3	Creating and Preparing the Source Files	45
Who Should Use This Book	4	Creating Packages for Embedded SQL	47
How to Use This Book	4	Precompiling	47
Conventions	7	Compiling and Linking	50
Related Publications.	8	Binding	51
Chapter 2. Coding a DB2 Application	9	Advantages of Deferred Binding	54
Prerequisites for Programming	9	DB2 Bind File Description Utility -	
DB2 Application Coding Overview.	10	db2bfd	54
Declaring and Initializing Variables	11	Application, Bind File, and Package	
Connecting to the Database Server	17	Relationships	55
Coding Transactions.	17	Timestamps	56
Ending the Program.	19	Rebinding	57
Implicitly Ending a Transaction	20	Chapter 4. Writing Static SQL Programs	59
Application Pseudocode Framework	21	Characteristics and Reasons for Using Static	
Designing an Application For DB2	22	SQL	60
Access to Data	23	Advantages of Static SQL	60
Data Value Control	27	Example: Static SQL Program	61
Data Relationship Control.	29	Coding SQL Statements to Retrieve and	
Application Logic at the Server	31	Manipulate Data	68
Supported SQL Statements	32	Using Host Variables	69
Authorization Considerations	32	Using Indicator Variables	73
Dynamic SQL	33	Selecting Multiple Rows Using a Cursor	78
Static SQL	33	Declaring and Using the Cursor.	79
Using APIs.	34	Cursors and Unit of Work	
Example	34	Considerations	80
Database Manager APIs Used in Embedded		Example: Cursor Program.	82
SQL or DB2 CLI Programs	35	Updating and Deleting Retrieved Data	90
Setting Up the Testing Environment	35	Updating Retrieved Data	90
Creating a Test Database	36	Deleting Retrieved Data	90
Creating Test Tables.	36	Types of Cursors	90
Generating Test Data	37	Example: OPENFTCH Program	91
Running, Testing and Debugging Your		Advanced Scrolling Techniques	101
Programs	39	Scrolling Through Data that has Already	
Prototyping Your SQL Statements	39	Been Retrieved	101
Part 2. Embedding SQL in Applications	41	Keeping a Copy of the Data	101
		Retrieving the Data a Second Time.	102
		Establishing a Position at the End of a	
		Table.	104
		Updating Previously Retrieved Data	104
		Example: UPDAT Program	104

Diagnostic Handling and the SQLCA Structure	117
Return Codes	117
SQLCODE and SQLSTATE	117
Token Truncation in SQLCA Structure	118
Handling Errors using the WHENEVER Statement	118
Exception, Signal, Interrupt Handler Considerations	119
Exit List Routine Considerations	120
Using GET ERROR MESSAGE in Example Programs	120
Chapter 5. Writing Dynamic SQL Programs	129
Why Use Dynamic SQL?	129
Dynamic SQL Support Statements	129
Comparing Dynamic SQL with Static SQL	130
Using PREPARE, DESCRIBE, FETCH and the SQLDA.	133
Declaring and Using Cursors.	134
Example: Dynamic SQL Program	136
Declaring the SQLDA	148
Preparing the Statement Using the Minimum SQLDA Structure	149
Allocating an SQLDA with Sufficient SQLVAR Entries	150
Describing the SELECT Statement	151
Acquiring Storage to Hold a Row	151
Processing the Cursor	152
Allocating an SQLDA Structure	152
Passing Data Using an SQLDA Structure	156
Processing Interactive SQL Statements	157
Saving SQL Requests from End Users	158
Example: ADHOC Program	159
Variable Input to Dynamic SQL	166
Using Parameter Markers	166
Example: VARINP Program	167
The DB2 Call Level Interface (CLI)	177
Comparing Embedded SQL and DB2 CLI	177
Advantages of Using DB2 CLI	178
Deciding on Embedded SQL or DB2 CLI	180

Part 3. Stored Procedures 183

Chapter 6. Stored Procedures	185
Stored Procedure Overview	185
Advantages of Stored Procedures	186
Writing Stored Procedures.	188

Client Application	189
Stored Procedure on the Server	190
Example OUT Stored Procedure.	200
Code Page Considerations.	208
C++ Consideration	208
Graphic Host Variable Considerations	209
Multisite Update Consideration	209
Building the Stored Procedure	209
Client Application	210
Stored Procedure.	210
NOT FENCED Stored Procedures	210
Returning Result Sets from Stored Procedures	212
Resolving Problems	214
Overview of Stored Procedure Builder	215
What is SPB?	215
Advantages of Using SPB	216

Part 4. Object-Relational Programming 219

Chapter 7. Using the Object-Relational Capabilities	221
Why Use the DB2 Object Extensions?	221
DB2 Approach to Supporting Objects	222
Combining UDTs, UDFs, and LOBs	223
Examples of Complex Applications.	223
Example: Defining the UDT and UDFs	223
Example: Exploiting LOB Function to Populate the Database	225
Example: Exploiting UDFs to Query Instances of UDTs	225

Chapter 8. Using Large Objects (LOBs)	227
What are LOBs?	227
Understanding Large Object Data Types (BLOB, CLOB, DBCLOB)	228
Understanding Large Object Locators	229
Example: Using a Locator to Work With a CLOB Value	231
How the Sample LOBLOC Program Works	231
C Sample: LOBLOC.SQC	232
COBOL Sample: LOBLOC.SQB	234
FORTRAN Sample: LOBLOC.SQF	237
Example: Deferring the Evaluation of a LOB Expression	239
How the Sample LOBEVAL Program Works	240

C Sample: LOBEVAL.SQC.	241	Defining Tables with UDTs	277
COBOL Sample: LOBEVAL.SQB.	243	Example: Sales	278
Indicator Variables and LOB Locators	246	Example: Application Forms	278
LOB File Reference Variables.	246	Manipulating UDTs	279
Example: Extracting a Document To a File	248	Examples of Manipulating UDTs	279
How the Sample LOBFILE Program		Example: Comparisons Between UDTs	279
Works	248	and Constants.	279
C Sample: LOBFILE.SQC	249	Example: Casting Between UDTs	280
COBOL Sample: LOBFILE.SQB	251	Example: Comparisons Involving UDTs	281
Example: Inserting Data Into a CLOB		Example: Sourced UDFs Involving UDTs	282
Column	253	Example: Assignments Involving UDTs	282
		Example: Assignments in Dynamic SQL	283
		Example: Assignments Involving	
		Different UDTs	283
		Example: Use of UDTs in UNION	284
Chapter 9. User-Defined Functions (UDFs)	255	Chapter 11. Writing User-Defined Functions (UDFs)	287
What are UDFs?	255	Description.	287
Why Use UDFs?	256	Interface between DB2 and a UDF	288
UDF Concepts	258	The Arguments Passed from DB2 to a	
Implementing UDFs.	260	UDF	288
Writing UDFs.	261	Summary of UDF Argument Use	301
Registering UDFs	261	How the SQL Data Types are Passed to a	
Examples of Registering UDFs	261	UDF	303
Example: Exponentiation	261	The UDF Include File: sqludf.h	311
Example: String Search.	262	Creating and Using Java User-Defined	
Example: BLOB String Search	263	Functions	312
Example: String Search over UDT	263	Coding a Java UDF	313
Example: External Function with UDT		Changing How a Java UDF Runs	314
Parameter	264	Table Function Execution Model for Java	315
Example: AVG over a UDT	264	Writing OLE Automation UDFs	317
Example: Counting	265	Creating and Registering OLE	
Example: Counting with an OLE		Automation UDFs	318
automation object	265	Object Instance and Scratchpad	
Example: Table Function Returning		Considerations	319
Document IDs.	266	How the SQL Data Types are Passed to	
Using UDFs	267	an OLE Automation UDF	319
Referring to Functions	267	Implementing OLE Automation UDFs in	
Examples of Function Invocations	268	BASIC and C++	320
Using Parameter Markers in Functions	269	OLE DB Table Functions	324
Using Qualified Function Reference	269	Creating an OLE DB Table Function	325
Using Unqualified Function Reference	270	Fully Qualified Rowset Names	327
Summary of Function References	271	Defining a Server Name for an OLE DB	
Chapter 10. User-defined Distinct Types (UDTs)	275	Provider	328
Why Use UDTs?	275	Defining a User Mapping	328
Defining a UDT	276	Supported OLE DB Data Types	329
Resolving Unqualified UDTs	276	Scratchpad Considerations	331
Examples of Using CREATE DISTINCT		Table Function Considerations	333
TYPE.	277	Table Function Error Processing.	334
Example: Money.	277		
Example: Job Application	277		

Scalar Function Error Processing	334
Using LOB Locators as UDF Parameters or Results	335
Scenarios for Using LOB Locators	339
Other Coding Considerations	340
Hints and Tips	340
UDF Restrictions and Caveats	342
Examples of UDF Code	344
Example: Integer Divide Operator	344
Example: Fold the CLOB, Find the Vowel	348
Example: Counter	353
Example: Weather Table Function	355
Example: Function using LOB locators	362
Example: Counter OLE Automation UDF in BASIC	365
Example: Counter OLE Automation UDF in C++	367
Debugging your UDF	371

Chapter 12. Using Triggers in an Active

DBMS	373
Why Use Triggers?	373
Benefits and Value of Triggers	374
Overview of a Trigger	375
Trigger Event	376
Set of Affected Rows	377
Trigger Granularity	377
Trigger Activation Time	378
Transition Variables	379
Transition Tables	380
Triggered Action	382
Triggered Action Condition	383
Triggered SQL Statements	383
Functions Within SQL Triggered Statement	383
Trigger Cascading	385
Interactions with Referential Constraints	385
Ordering of Multiple Triggers	385
Synergy Between Triggers, Constraints, UDTs, UDFs, and LOBs	386
Extracting Information	386
Preventing Operations on Tables	387
Defining Business Rules	388
Defining Actions	388

Part 5. DB2 Programming Considerations 391

Chapter 13. Programming in Complex Environments	393
National Language Support Considerations	394
Collating Sequences	394
Deriving Code Page Values	401
Deriving Locales in Application Programs	402
Programming Considerations	403
Unicode/UCS-2 and UTF-8 Support in DB2 UDB	410
Running CLI/ODBC/JDBC/SQLJ Programs in a DBCS Environment	418
Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations	419
Mixed EUC and Double-Byte Client and Database Considerations	421
Considerations for Traditional Chinese Users.	422
Developing Japanese or Traditional Chinese EUC Applications	422
Developing for Mixed Code Set Environments	424
Applications Connected to a Unicode (UCS-2) Database	433
Considerations for Multisite Updates	433
Remote Unit of Work	434
Multisite Update	434
Accessing Host or AS/400 Servers	441
Multiple Thread Database Access	441
Recommendations for Using Multiple Threads	443
Multithreaded UNIX Applications Working with Code Page and Country Code	443
Potential Pitfalls when Using Multiple Threads	444
Concurrent Transactions	446
Potential Pitfalls when Using Concurrent Transactions	446
X/Open XA Interface Programming Considerations	448
Application Linkage.	451
Working with Large Volumes of Data Across a Network	451

Chapter 14. Programming Considerations in a Partitioned Environment	453
Improving Performance	453
Using FOR READ ONLY Cursors	453
Using Directed DSS and Local Bypass	453

Using Buffered Inserts	455	Trigraph Sequences	491
Example: Extracting Large Volume of Data (largevol.c)	461	C++ Type Decoration Consideration	492
Creating a Test Environment	466	Input and Output Files.	492
Error-Handling Considerations	467	Include Files	493
Severe Errors	467	Including Files	496
Merged Multiple SQLCA Structures	468	Embedding SQL Statements	497
Identifying the Partition that Returned the Error	469	Host Variables.	498
Debugging.	469	Naming Host Variables.	498
Diagnosing a Looping or Suspended application.	469	Declaring Host Variables	499
		Indicator Variables	503
		Graphic Host Variable Declarations in C or C++	503
		LOB Data Declarations in C or C++	506
		LOB Locator Declarations in C or C++	508
		File Reference Declarations in C or C++	509
		Initializing Host Variables.	510
		C Macro Expansion	510
		Host Structure Support.	511
		Indicator Tables	513
		Null-terminated Strings	514
		Pointer Data Types	516
		Using Class Data Members as Host Variables	517
		Using Qualification and Member Operators	518
		Handling Graphic Host Variables	519
		Japanese or Traditional Chinese EUC, and UCS-2 Considerations	524
		Supported SQL Data Types	525
		FOR BIT DATA	530
		SQLSTATE and SQLCODE Variables	530
Chapter 15. Writing Programs for DB2 Federated Systems	471	Chapter 17. Programming in Java	533
Introduction to DB2 Federated Systems	471	Programming Considerations	533
Accessing Data Source Tables and Views	472	Comparison of SQLJ to JDBC	533
Working with Nicknames	472	Advantages Over Other Languages	534
Using Isolation Levels to Maintain Data Integrity.	476	SQL Security	534
Working with Data Type Mappings	477	Source and Output Files	534
How DB2 Determines What Data Types to Define Locally.	477	Java Class Libraries	535
Default Data Type Mappings.	477	Java Packages.	535
How You Can Override Default Type Mappings and Create New Ones	478	Supported SQL Data Types	535
Using Distributed Requests to Query Data Sources	479	SQLSTATE and SQLCODE Values	537
Coding Distributed Requests.	479	DB2 Trace Facilities	538
Using Server Options to Facilitate Optimization	481	Creating Java Applications and Applets	538
Invoking Data Source Functions.	482	JDBC Programming.	540
Enabling DB2 to Invoke Data Source Functions	482	How the DB2Appl Program Works.	541
Reducing the Overhead of Invoking a Function	483	Distributing a JDBC Application	543
Specifying Function Names in the CREATE FUNCTION MAPPING Statement	484	Distributing and Running a JDBC Applet	543
Discontinuing Function Mappings	484	JDBC 2.0	544
Using Pass-Through to Query Data Sources Directly	485	SQLJ Programming	545
SQL Processing in Pass-Through Sessions	485		
Considerations and Restrictions	486		
Part 6. Language Considerations 489			
Chapter 16. Programming in C and C++ 491			
Programming Considerations	491		
Language Restrictions	491		

DB2 SQLJ Support	546	Including Files	595
Embedding SQL Statements	548	Embedding SQL Statements	595
Host Variables.	554	Host Variables.	597
Calls to Stored Procedures and Functions	554	Naming Host Variables.	597
Compiling and Running SQLJ Programs	555	Declaring Host Variables	597
SQLJ Precompile Options	556	Indicator Variables	600
Stored Procedures and UDFs.	557	LOB Declarations in FORTRAN	600
Where to Put Java Classes.	558	LOB Locator Declarations in FORTRAN	601
Debugging Stored Procedures	559	File Reference Declarations in FORTRAN	601
Java Stored Procedures and UDFs	562	Supported SQL Data Types	602
Using LOBs and Graphical Objects With		SQLSTATE and SQLCODE Variables	604
JDBC 1.22	566	Considerations for Multi-byte Character Sets	605
JDBC and SQLJ Interoperability	567	Japanese or Traditional Chinese EUC, and	
Session Sharing	567	UCS-2 Considerations	605
Connection Resource Management	567		
Chapter 18. Programming in COBOL	569	Chapter 20. Programming in REXX	607
Programming Considerations	569	Programming Considerations	607
Language Restrictions	569	Language Restrictions	608
Input and Output Files.	569	Registering SQLEXEC, SQLDBS and	
Include Files	570	SQLDB2.	608
Embedding SQL Statements	573	Embedding SQL Statements	609
Host Variables.	575	Host Variables.	611
Naming Host Variables.	575	Naming Host Variables.	611
Declaring Host Variables	575	Referencing Host Variables	611
Indicator Variables	579	Indicator Variables in REXX	612
LOB Declarations in COBOL	579	Predefined REXX Variables	612
LOB Locator Declarations in COBOL	581	LOB Host Variables in REXX.	614
File Reference Declarations in COBOL	581	LOB Locator Declarations in REXX.	615
Host Structure Support.	582	LOB File Reference Declarations in REXX	615
Indicator Tables	584	Clearing LOB Host Variables.	617
Using REDEFINES in COBOL Group		Supported SQL Data Types	617
Data Items	585	Using Cursors in REXX	619
Using BINARY/COMP-4 COBOL Data		Execution Requirements for REXX	619
Types	585	Bind Files for REXX.	620
Supported SQL Data Types	586	API Syntax.	621
FOR BIT DATA	589	REXX Stored Procedures	623
SQLSTATE and SQLCODE Variables	589	Calling Stored Procedures.	623
Japanese or Traditional Chinese EUC, and		Japanese or Traditional Chinese EUC	
UCS-2 Considerations	590	Considerations	625
Object Oriented COBOL	590		
Chapter 19. Programming in FORTRAN	591	Part 7. Appendixes	627
Programming Considerations	591	Appendix A. Supported SQL Statements	
Language Restrictions	591	(DB2 Universal Database)	629
Call by Reference	591		
Debugging and Comment Lines.	592	Appendix B. Sample Programs	633
Precompiling Considerations.	592	DB2 API Non-Embedded SQL Samples	638
Input and Output Files.	592	DB2 API Embedded SQL Samples	640
Include Files	592	Embedded SQL Samples With No DB2 APIs	642

User-Defined Function Samples	645	Package Attributes	684
DB2 Call Level Interface Samples	645	C Null-terminated Strings	685
Java Samples	648	Standalone SQLCODE and SQLSTATE	685
ADO, RDO, and MTS Samples	650	Defining a Sort Order	686
Object Linking and Embedding Samples	651	Managing Referential Integrity	686
Command Line Processor Samples	652	Locking	687
Log Management User Exit Samples	653	Differences in SQLCODEs and SQLSTATes	687
Appendix C. DB2DARI and DB2GENERAL		Using System Catalogs	687
Stored Procedures and UDFs	655	Numeric Conversion Overflows on	
DB2DARI Stored Procedures	655	Retrieval Assignments	688
Using the SQLDA in a Client Application	655	Isolation Levels	688
Using Host Variables in a DB2DARI		Stored Procedures	689
Client	656	NOT ATOMIC Compound SQL	690
Using the SQLDA in a Stored Procedure	656	Multisite Update with DB2 Connect	691
Summary of Data Structure Usage	657	Host or AS/400 Server SQL Statements	
Input/Output SQLDA and SQLCA		Supported by DB2 Connect	692
Structures	658	Host or AS/400 Server SQL Statements	
DB2GENERAL UDFs and Stored Procedures	659	Rejected by DB2 Connect	693
Supported SQL Data Types	659	Appendix E. Country Code and Code	
Classes for Java Stored Procedures and		Page Support	695
UDFs	661	Appendix F. Simulating EBCDIC Binary	
NOT FENCED Stored Procedures	667	Collation	711
Example Input-SQLDA Programs	668	Appendix G. How the DB2 Library Is	
How the Example Input-SQLDA Client		Structured.	717
Application Works	669	Completing Tasks with SmartGuides	717
FORTRAN UNIX Example: INPCLISQF	671	Accessing Online Help	718
How the Example Input-SQLDA Stored		DB2 Information – Hardcopy and Online	720
Procedure Works	675	Viewing Online Information	727
REXX OS/2 Example: INPSRV.CMD	676	Accessing Information with the	
Appendix D. Programming in a Host or		Information Center	728
AS/400 Environment	679	Setting Up a Document Server	729
Using Data Definition Language (DDL)	681	Searching Online Information	730
Using Data Manipulation Language (DML)	681	Printing the PostScript Books.	730
Numeric Data Types	681	Ordering the Printed Books	732
Mixed-Byte Data	681	Appendix H. Notices	733
Long Fields	682	Trademarks	734
Large Object (LOB) Data Type	682	Trademarks of Other Companies	734
User Defined Types (UDTs)	682	Index	737
ROWID Data Type	682	Contacting IBM	763
64-bit Integer (BIGINT) data type	682		
Using Data Control Language (DCL)	682		
Connecting and Disconnecting	682		
Precompiling	683		
Blocking	684		

Part 1. DB2 Application Development Concepts

Chapter 1. Getting Started with DB2 Application Development

About This Book	3	Conventions	7
Who Should Use This Book	4	Related Publications.	8
How to Use This Book	4		

About This Book

This book discusses how to design and code application programs that access DB2 databases and application servers. It presents detailed information on the use of Structured Query Language (SQL) in supported host language programs. For information unique to your specific operating system, see the *Application Building Guide*.

You can access data with:

- SQL statements embedded in a host language, including embedded SQL for Java (SQLJ)
- dynamic APIs including Java Database Connectivity (JDBC), Perl DBI, and DB2 Call Level Interface (DB2 CLI)

This book discusses all these ways to access data except DB2 CLI, which is discussed in the *CLI Guide and Reference*. JDBC, SQLJ, and DB2 CLI provide some data access capabilities that are not available through embedded SQL. These capabilities include scrollable cursors and stored procedures that return multiple result sets. See the discussion in “Access to Data” on page 23 to help you decide which data access method to use.

To effectively utilize the information in this book to design, write, and test your DB2 application programs, you need to refer to the *SQL Reference* along with this book. If you are using the DB2 Call Level Interface (CLI) or Open Database Connectivity (ODBC) interface in your applications to access DB2 databases, refer to the *CLI Guide and Reference*. To perform database manager administration functions using the DB2 administration APIs in your application programs, refer to the *Administrative API Reference*.

You can also develop applications where one part of the application runs on the client and another part runs on the server. Version 6 of DB2 introduces support for stored procedures with enhanced portability and scalability across platforms. Stored procedures are discussed in “Chapter 6. Stored Procedures” on page 185.

You can use object-based extensions to DB2 to make your DB2 application programs more powerful, flexible, and active than traditional DB2 applications. The extensions include large objects (*LOBs*), user-defined distinct types (*UDTs*), user-defined functions (*UDFs*), and triggers. These features of DB2 are described in:

- “Chapter 7. Using the Object-Relational Capabilities” on page 221
- “Chapter 8. Using Large Objects (LOBs)” on page 227
- “Chapter 9. User-Defined Functions (UDFs)” on page 255
- “Chapter 10. User-defined Distinct Types (UDTs)” on page 275
- “Chapter 11. Writing User-Defined Functions (UDFs)” on page 287
- “Chapter 12. Using Triggers in an Active DBMS” on page 373

References to DB2 in this book should be understood to mean the DB2 Universal Database product. References to DB2 on other platforms use a specific product name, such as DB2 Universal Database for AS/400.

Who Should Use This Book

This book is intended for programmers who are experienced with SQL and with one or more of the supported programming languages.

How to Use This Book

This book is organized, by task, into the following parts, chapters, and appendices:

- Part 1. DB2 Application Development Concepts contains information you need to use this book and an overview of the methods you can use to develop applications for DB2 Universal Database.
 - Chapter 1. Getting Started with DB2 Application Development describes the structure of this book and the conventions used in it.
 - Chapter 2. Coding a DB2 Application introduces the overall application development process using DB2. It discusses and compares the important application design issues you need to consider prior to coding your applications. This chapter concludes with information to help you set up a test environment where you can begin to develop your applications.
- Part 2. Embedding SQL in Applications describes how to embed static and dynamic SQL in your applications. This information includes a description of the utilities that you can use to help create your embedded SQL applications.

- Chapter 4. Writing Static SQL Programs discusses the details of coding your DB2 embedded SQL application using static SQL statements. It contains detailed guidelines and considerations for using static SQL.
- Chapter 5. Writing Dynamic SQL Programs discusses the details of coding your DB2 embedded SQL application using dynamic SQL statements. It contains detailed guidelines and considerations for using dynamic SQL.
- Part 3. Stored Procedures discusses how to use stored procedures to improve the performance of database applications that run in client/server environments.
 - Chapter 6. Stored Procedures describes how to develop stored procedures and the client applications that call stored procedures.
- Part 4. Object-Relational Programming describes how to use the object-relational support provided by DB2. This information includes an introduction to and detailed instructions on how to use large objects, user-defined functions, user-defined distinct types, and triggers.
 - Chapter 7. Using the Object-Relational Capabilities introduces the object-oriented capabilities of DB2. It explains how to extend your traditional application to one that takes advantage of DB2 capabilities such as large objects, user-defined functions, and user-defined distinct types in an object-oriented context.
 - Chapter 8. Using Large Objects (LOBs) describes how to define and use data types that can store data objects as binary or text strings of up to two gigabytes in size. It also explains how to efficiently use LOBs in a networked environment.
 - Chapter 9. User-Defined Functions (UDFs) describes how to write your own extensions to SQL. It explains how to use UDFs to express the behavior of your data objects.
 - Chapter 10. User-defined Distinct Types (UDTs) describes how to create and use your own data types in applications. It explains how to use UDTs as a foundation for object-oriented extensions to the built-in data types.
 - Chapter 11. Writing User-Defined Functions (UDFs) describes how to write user-defined functions that extend your DB2 applications. Topics include the details of writing a user-defined function, programming considerations for user-defined functions, and several examples that show you how to exploit this important capability. In addition, this chapter describes user-defined table functions, OLE DB table functions, and OLE automation UDFs.
 - Chapter 12. Using Triggers in an Active DBMS describes how to use triggers to encapsulate and enforce business rules within all of your database applications.

- Part 5. DB2 Programming Considerations contains information on special application development considerations.
 - Chapter 13. Programming in Complex Environments discusses advanced programming topics such as national language support, dealing with Extended UNIX® Code (EUC) code pages for databases and applications, accessing multiple databases within a unit of work, and multi-threaded applications.
 - Chapter 14. Programming Considerations in a Partitioned Environment describes programming considerations if you are developing applications that run in a partitioned environment.
 - Chapter 15. Writing Programs for DB2 Federated Systems describes how to create applications that transparently access data from DB2 family and Oracle data sources through a federated server.
- Part 6. Language Considerations contains specific information about the programming languages that DB2 supports.
 - Chapter 16. Programming in C and C++ discusses host language specific information concerning database applications written in C and C++.
 - Chapter 17. Programming in Java discusses host language specific information concerning database applications written in Java using JDBC or SQLJ.
 - Chapter 18. Programming in COBOL discusses host language specific information concerning database applications written in COBOL.
 - Chapter 19. Programming in FORTRAN discusses host language specific information concerning database applications written in FORTRAN.
 - Chapter 20. Programming in REXX discusses host language specific information concerning database applications written in REXX.
- The Appendices contain supplemental information to which you may need to refer when developing DB2 applications.
 - Appendix A. Supported SQL Statements (DB2 Universal Database) lists the SQL statements supported by DB2 Universal Database.
 - Appendix B. Sample Programs contains information on supplied sample programs for supported host languages and describes how they work.
 - Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs contains information you can use to create stored procedures and UDFs that are compatible with previous versions of DB2 Universal Database.
 - Appendix D. Programming in a Host or AS/400 Environment describes programming considerations for DB2 Connect if you access host or AS/400 database servers in your applications in a distributed environment.
 - Appendix E. Country Code and Code Page Support lists the languages and code sets supported by DB2 Universal Database servers.

- Appendix F. Simulating EBCDIC Binary Collation describes how to collate DB2 character strings according to an EBCDIC, or user-defined, collating sequence.
- Appendix G. How the DB2 Library Is Structured shows you where you can get more information for the DB2 Universal Database product.

Conventions

This book uses the following conventions:

Directories and Paths

This book uses the UNIX convention for delimiting directories, for example: `sqllib/samples/java`. You can convert these paths to Windows 32-bit operating system and OS/2 paths by changing the `/` to a `\` and prepending the appropriate installation drive and directory.

Italics Indicates one of the following:

- Introduction of a new term
- Variable names or values that are supplied by the user
- Reference to another source of information, for example, a book or CD-ROM
- General emphasis

UPPERCASE

Indicates one of the following:

- API names
- Abbreviations
- Database manager data types
- Function names
- SQL statements

Example

Indicates one of the following:

- Coding examples and code fragments
- Examples of output, similar to what is displayed by the system
- Examples of specific data values
- Examples of system messages
- File and directory names
- Information that you are instructed to type
- Java method names

Bold Bold text emphasizes a point.

Related Publications

The following manuals describe how to develop applications for international use and for specific countries:

Form Number	Book Title
SE09-8001-03	<i>National Language Design Guide, Volume 1</i>
SE09-8002-03	<i>NLS Reference Manual, Release 4</i>

Chapter 2. Coding a DB2 Application

Prerequisites for Programming	9	Data Types	27
DB2 Application Coding Overview.	10	Unique Constraints	28
Declaring and Initializing Variables	11	Table Check Constraints	28
Declaring Variables that Interact with		Referential Integrity Constraints.	28
the Database Manager	11	Views with Check Option	29
Handling Errors and Warnings	14	Application Logic and Program	
Using Additional Nonexecutable		Variable Types	29
Statements	16	Data Relationship Control.	29
Connecting to the Database Server	17	Referential Integrity Constraints.	30
Coding Transactions.	17	Triggers	30
Beginning a Transaction	18	Application Logic	31
Ending a Transaction	18	Application Logic at the Server	31
Ending the Program.	19	Stored Procedures	31
Implicitly Ending a Transaction	20	User-Defined Functions	31
On Most Supported Operating		Triggers	32
Systems	20	Supported SQL Statements	32
On Windows 32-bit Operating		Authorization Considerations	32
Systems	20	Dynamic SQL	33
When Using the DB2 Context APIs	21	Static SQL	33
Application Pseudocode Framework	21	Using APIs.	34
Designing an Application For DB2	22	Example	34
Access to Data	23	Database Manager APIs Used in Embedded	
Embedded SQL	24	SQL or DB2 CLI Programs	35
DB2 Call Level Interface (DB2 CLI)		Setting Up the Testing Environment	35
and Open Database Connectivity		Creating a Test Database	36
(ODBC)	25	Creating Test Tables.	36
Microsoft Specifications	26	Generating Test Data	37
Perl DBI.	26	Running, Testing and Debugging Your	
Query Products	27	Programs	39
Data Value Control	27	Prototyping Your SQL Statements	39

Prerequisites for Programming

This chapter presents a model of the logical parts of a DB2 application and discusses the individual strengths of the supported DB2 programming APIs. Programmers who are new to developing a DB2 application should read the entire chapter closely.

The application development process described in this book assumes that you have established the appropriate operating environment. This means that the following are properly installed and configured:

- A supported compiler or interpreter for developing your applications.
- DB2 Universal Database, either locally or remotely.

- DB2 Software Developer's Kit.

For details on how to accomplish these tasks, see the *Application Building Guide* and *Quick Beginnings* books for your operating environment.

You can *develop* applications at a server, or on any client, that has the DB2 Software Developer's Kit (DB2 SDK) installed. You can *run* applications with either the server, the DB2 Run-Time Client, or the DB2 Administrative Client. You can also develop Java JDBC programs on one of these clients, provided that you install the "Java Enablement" component when you install the client. That means you can execute any DB2 application on these clients. However, unless you also install the DB2 SDK with these clients, you can only develop JDBC applications on them.

DB2 supports the C, C++, Java (SQLJ), COBOL, and FORTRAN programming languages through its precompilers. In addition, DB2 provides support for the REXX and Java (JDBC) dynamically interpreted languages. For information on the specific precompilers provided by DB2, and the languages supported on your platform, refer to the *Application Building Guide*.

Note: The FORTRAN and REXX interfaces support only the function up to and including DB2 Version 5.2.

DB2 provides a sample database which you require when running the supplied sample programs. For information about the sample database and its contents, refer to the *SQL Reference*.

DB2 Application Coding Overview

A DB2 application program consists of several parts:

1. Declaring and initializing variables
2. Connecting to the database
3. Performing one or more *transactions*
4. Disconnecting from the database
5. Ending the program

A *transaction* is a set of database operations that must conclude successfully before being committed to the database. With embedded SQL, a transaction begins implicitly and ends when the application executes either a COMMIT or ROLLBACK statement. An example of a transaction is the entry of a customer's deposit, and the updating of the customer's balance.

Certain SQL statements must appear at the beginning and end of the program to handle the transition from the host language to the embedded SQL statements.

The beginning of every program must contain:

- Declarations of all variables and data structures that the database manager uses to interact with the host program
- SQL statements that provide for error handling by setting up the SQL Communications Area (SQLCA)

Note that DB2 applications written in Java throw an `SQLException`, which you handle in a catch block, rather than using the SQLCA.

The body of every program contains the SQL statements that access and manage data. These statements constitute transactions. Transactions must include the following statements:

- The `CONNECT` statement, which establishes a connection to a database server
- One or more:
 - Data manipulation statements (for example, the `SELECT` statement)
 - Data definition statements (for example, the `CREATE` statement)
 - Data control statements (for example, the `GRANT` statement)
- Either the `COMMIT` or `ROLLBACK` statement to end the transaction

The end of the application program typically contains SQL statements that:

- Release the program's connection to the database server
- Clean up any resource

Declaring and Initializing Variables

To code a DB2 application, you must first declare:

- the variables that interact with the database manager
- the SQLCA, if applicable

Declaring Variables that Interact with the Database Manager

All variables that interact with the database manager must be declared in an SQL declare section. You must code an SQL declare section with the following structure:

1. the SQL statement `BEGIN DECLARE SECTION`
2. a group of one or more variable declarations
3. the SQL statement `END DECLARE SECTION`

Host program variables declared in an SQL declare section are called *host variables*. You can use host variables in *host-variable* references in SQL statements. *Host-variable* is a tag used in syntax diagrams in the *SQL Reference*. A program may contain multiple SQL declare sections.

The attributes of each host variable depend on how the variable is used in the SQL statement. For example, variables that receive data from or store data in DB2 tables must have data type and length attributes compatible with the column being accessed. To determine the data type for each variable, you must be familiar with DB2 data types, which are explained in “Data Types” on page 75.

Declaring Variables that Represent SQL Objects: For DB2 Version 6, the names of tables, aliases, views, and correlations have a maximum length of 128 bytes. Column names have a maximum length of 30 bytes. Future releases of DB2 may increase the lengths of column names and other identifiers of SQL objects up to 128 bytes. If you declare variables that represent SQL objects with less than 128 byte lengths, future increases in SQL object identifier lengths may affect the stability of your applications. For example, if you declare the variable `char[9] schema_name` in a C++ application to hold a schema name, your application functions properly for the allowed schema names in DB2 Version 6, which have a maximum length of 8 bytes.

```
char[9] schema_name; /* holds null-delimited schema name of up to 8 bytes;
works for Version 6 but may truncate schema names in future releases */
```

However, if you migrate the database to a future version of DB2 which accepts schema names with a maximum length of 128 bytes, your application cannot differentiate between the schema names LONGSCHEMA1 and LONGSCHEMA2. The database manager truncates the schema names to their 8-byte limit of LONGSCHE, and any statement in your application that depends on differentiating the schema names fails. To increase the longevity of your application, declare the schema name variable with a 128-byte length as follows:

```
char[129] schema_name; /* holds null-delimited schema name of up to 128 bytes
good for DB2 Version 6 and beyond */
```

To improve the future operation of your application, consider declaring all of the variables in your applications that represent SQL object names with lengths of 128 bytes. You must weigh the advantage of improved compatibility against the increased system resources that longer variables require.

To ease the use of this coding practice and increase the clarity of your C/C++ application code clear, consider using C macro expansion to declare the lengths of these SQL object identifiers. Since the include file `sql.h` declares

SQL_MAX_IDENT to be 128, you can easily declare SQL object identifiers with the SQL_MAX_IDENT macro. For example:

```
#include <sql.h>
char[SQL_MAX_IDENT+1] schema_name;
char[SQL_MAX_IDENT+1] table_name;
char[SQL_MAX_IDENT+1] employee_column;
char[SQL_MAX_IDENT+1] manager_column;
```

For more information on C macro expansion, see “C Macro Expansion” on page 510.

Relating Host Variables to an SQL Statement: You can use host variables to receive data from the database manager or to transfer data to it from the host program. Host variables that receive data from the database manager are *output host variables*, while those that transfer data to it from the host program are *input host variables*.

Consider the following SELECT INTO statement:

```
SELECT HIREDATE, EDLEVEL
  INTO :hdate, :lvl
  FROM EMPLOYEE
 WHERE EMPNO = :idno
```

It contains two output host variables, hdate and lvl, and one input host variable, idno. The database manager uses the data stored in the host variable idno to determine the EMPNO of the row that is retrieved from the EMPLOYEE table. If the database manager finds a row that meets the search criteria, hdate and lvl receive the data stored in the columns HIREDATE and EDLEVEL, respectively. This statement illustrates an interaction between the host program and the database manager using columns of the EMPLOYEE table.

Each column of a table is assigned a data type in the CREATE TABLE definition. You must relate this data type to the host language data type defined in the *Supported SQL Data Types* section of each language-specific chapter in this document. For example, the INTEGER data type is a 32-bit signed integer. This is equivalent to the following data description entries in each of the host languages, respectively:

C/C++:

```
long variable_name;
```

Java: int variable_name;

COBOL:

```
01 variable-name PICTURE S9(9) COMPUTATIONAL-5.
```

FORTRAN:

```
INTEGER*4 variable_name
```

For the list of supported SQL data types and the corresponding host language data types, see the following:

- for C/C++, “Supported SQL Data Types” on page 525
- for Java, “Supported SQL Data Types” on page 535
- for COBOL, “Supported SQL Data Types” on page 586
- for FORTRAN, “Supported SQL Data Types” on page 602
- for REXX, “Supported SQL Data Types” on page 617

In order to determine exactly how to define the host variable for use with a column, you need to find out the SQL data type for that column. Do this by querying the system catalog, which is a set of views containing information about all tables created in the database. The *SQL Reference* describes this catalog.

After you have determined the data types, you can refer to the conversion charts in the host language chapters and code the appropriate declarations. The Declaration Generator utility (db2dc1gn) is also available for generating the appropriate declarations for a given table in a database. For more information on db2dc1gn, see “Declaration Generator - db2dc1gn” on page 71 and refer to the *Command Reference*.

Table 4 on page 72 shows examples of declarations in the supported host languages. Note that REXX applications do not need to declare host variables except for LOB locators and file reference variables. The contents of the variable determine other host variable data types and sizes at run time.

Table 4 also shows the BEGIN and END DECLARE SECTION statements. Observe how the delimiters for SQL statements differ for each language. For the exact rules of placement, continuation, and delimiting of these statements, see the language-specific chapters of this book.

Handling Errors and Warnings

The SQL Communications Area (SQLCA) is discussed in detail later in this chapter. This section presents an overview. To declare the SQLCA, code the INCLUDE SQLCA statement in your program.

For C or C++ applications use:

```
EXEC SQL INCLUDE SQLCA;
```

For Java applications: You do not explicitly use the SQLCA in Java. Instead, use the SQLException instance methods to get the SQLSTATE and SQLCODE values. See “SQLSTATE and SQLCODE Values” on page 537 for more details.

For COBOL applications use:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

For FORTRAN applications use:

```
EXEC SQL INCLUDE SQLCA
```

When you preprocess your program, the database manager inserts host language variable declarations in place of the INCLUDE SQLCA statement. The system communicates with your program using the variables for warning flags, error codes, and diagnostic information.

After executing each SQL statement, the system returns a return code in both SQLCODE and SQLSTATE. SQLCODE is an integer value that summarizes the execution of the statement, and SQLSTATE is a character field that provides common error codes across IBM's relational database products. SQLSTATE also conforms to the ISO/ANS SQL92 and FIPS 127-2 standard.

Note: FIPS 127-2 refers to *Federal Information Processing Standards Publication 127-2 for Database Language SQL*. ISO/ANS SQL92 refers to *American National Standard Database Language SQL X3.135-1992* and *International Standard ISO/IEC 9075:1992, Database Language SQL*.

Note that if SQLCODE is less than 0, it means an error has occurred and the statement has not been processed. If the SQLCODE is greater than 0, it means a warning has been issued, but the statement is still processed. See the *Message Reference* for a listing of SQLCODE and SQLSTATE error conditions.

If you want the system to control error checking after each SQL statement, use the WHENEVER statement.

Note: Embedded SQL for Java (SQLJ) applications cannot use the WHENEVER statement. Use the SQLException methods described in “SQLSTATE and SQLCODE Values” on page 537 to handle errors returned by SQL statements.

The following WHENEVER statement indicates to the system what to do when it encounters a negative SQLCODE:

```
WHENEVER SQLERROR GO TO errchk
```

That is, whenever an SQL error occurs, program control is transferred to code that follows the label, such as errchk. This code should include logic to analyze the error indicators in the SQLCA. Depending upon the ERRCHK definition, action may be taken to execute the next sequential program instruction, to perform some special functions, or as in most situations, to roll back the current *transaction* and terminate the program. See “Coding

Transactions” on page 17 for more information on a transaction and “Diagnostic Handling and the SQLCA Structure” on page 117 for more information about how to control error checking in your application program.

Exercise caution when using the WHENEVER SQLERROR statement. If your application’s error handling code contains SQL statements, and if these statements result in an error while processing the original error, your application may enter an infinite loop. This situation is difficult to troubleshoot. The first statement in the destination of a WHENEVER SQLERROR should be WHENEVER SQLERROR CONTINUE. This statement resets the error handler. After this statement, you can safely use SQL statements.

For a DB2 application written in C or C++, if the application is made up of multiple source files, only one of the files should include the EXEC SQL INCLUDE SQLCA statement to avoid multiple definitions of the SQLCA. The remaining source files should use the following lines:

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

If your application must be compliant with the ISO/ANS SQL92 or FIPS 127-2 standard, do not use the above statements or the INCLUDE SQLCA statement. For more information on the ISO/ANS SQL92 and FIPS 127-2 standards, see “Definition of FIPS 127-2 and ISO/ANS SQL92” on page 15. For the alternative to coding the above statements, see the following:

- For C or C++ applications, see “SQLSTATE and SQLCODE Variables” on page 530
- For COBOL applications, “SQLSTATE and SQLCODE Variables” on page 589
- For FORTRAN applications, “SQLSTATE and SQLCODE Variables” on page 604

Using Additional Nonexecutable Statements

Generally, other *nonexecutable* SQL statements are also part of this section of the program. Both the *SQL Reference* and subsequent chapters of this manual discuss nonexecutable statements. Examples of nonexecutable statements are:

- INCLUDE text-file-name
- INCLUDE SQLDA
- DECLARE CURSOR

Connecting to the Database Server

Your program must establish a connection to the target database server before it can run any executable SQL statements. This connection identifies both the authorization ID of the user who is running the program, and the name of the database server on which the program is run. Generally, your application process can only connect to one database server at a time. This server is called the *current server*. However, your application can connect to multiple database servers within a multisite update. In this case, only one server can be the current server. For more information on multisite updates, see “Multisite Update” on page 434.

Your program can establish a connection to a database server either:

- explicitly, using the `CONNECT` statement
- implicitly, connecting to the default database server
- for Java applications, through a `Connection` instance

Refer to the *SQL Reference* for a discussion of connection states and how to use the `CONNECT` statement. Upon initialization, the application requester establishes a default database server. If implicit connects are enabled, application processes started after initialization connect implicitly to the default database server. It is good practice to use the `CONNECT` statement as the first SQL statement executed by an application program. This avoids accidentally executing SQL statements against the default database.

After the connection has been established, your program can issue SQL statements that:

- Manipulate data
- Define and maintain database objects
- Initiate control operations, such as granting user authority, or committing changes to the database

A connection lasts until a `CONNECT RESET`, `CONNECT TO`, or `DISCONNECT` statement is issued. In a multisite update environment, a connection also lasts until a `DB2 RELEASE` then `DB2 COMMIT` is issued. A `CONNECT TO` statement does not terminate a connection when using multisite update (see “Multisite Update” on page 434).

Coding Transactions

A transaction is a sequence of SQL statements (possibly with intervening host language code) that the database manager treats as a whole. An alternative term that is often used for transaction is *unit of work*.

To ensure the consistency of data at the transaction level, the system makes sure that either *all* operations within a transaction are completed, or *none* are completed. Suppose, for example, that the program is supposed to deduct money from one account and add it to another. If you place both of these updates in a single transaction, and a system failure occurs while they are in progress, then when you restart the system, the database manager automatically restores the data to the state it was in before the transaction began. If a program error occurs, the database manager restores all changes made by the statement in error. The database manager will not undo work performed in the transaction prior to execution of the statement in error, unless you specifically roll it back.

You can code one or more transactions within a single application program, and it is possible to access more than one database from within a single transaction. A transaction that accesses more than one database is called a multisite update. For information on these topics, see “Remote Unit of Work” on page 434 and “Multisite Update” on page 434.

Beginning a Transaction

A transaction begins implicitly with the first *executable* SQL statement and ends with either a COMMIT or a ROLLBACK statement, or when the program ends.

In contrast, the following six statements do **not** start a transaction because they are not executable statements:

BEGIN DECLARE SECTION	INCLUDE SQLCA
END DECLARE SECTION	INCLUDE SQLDA
DECLARE CURSOR	WHenever

An executable SQL statement always occurs within a transaction. If a program contains an executable SQL statement after a transaction ends, it automatically starts a new transaction.

Ending a Transaction

To end a transaction, you can use either:

- The COMMIT statement to save its changes
- The ROLLBACK statement to ensure that these changes are not saved

Using the COMMIT Statement: This statement ends the current transaction. It makes the database changes performed during the current transaction visible to other processes.

You should commit changes as soon as application requirements permit. In particular, write your programs so that uncommitted changes are not held

while waiting for input from a terminal, as this can result in database resources being held for a long time. Holding these resources prevents other applications that need these resources from running.

The COMMIT statement has no effect on the contents of host variables.

Your application programs should explicitly end any transactions prior to terminating. If you do not end transactions explicitly, DB2 automatically commits all the changes made during the program's pending transaction when the program ends successfully, except on Windows 32-bit operating systems. DB2 rolls back the changes under the following conditions:

- A log full condition
- Any other system condition that causes database manager processing to end

On Windows 32-bit operating systems, if you do not explicitly commit the transaction, the database manager always rolls back the changes.

For more information about program termination, see “Ending the Program” and “Diagnostic Handling and the SQLCA Structure” on page 117.

Using the ROLLBACK Statement: This statement ends the current transaction, and restores the data to the state it was in prior to beginning the transaction.

The ROLLBACK statement has no effect on the contents of host variables.

If you use a ROLLBACK statement in a routine that was entered because of an error or warning and you use the SQL WHENEVER statement, then you should specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK. This avoids a program loop if the ROLLBACK fails with an error or warning.

In the event of a severe error, you will receive a message indicating that you cannot issue a ROLLBACK statement. Do not issue a ROLLBACK statement if a severe error occurs such as the loss of communications between the client and server applications, or if the database gets corrupted. After a severe error, the only statement you can issue is a CONNECT statement.

Ending the Program

To properly end your program:

1. End the current transaction (if one is in progress) by explicitly issuing either a COMMIT statement or a ROLLBACK statement.

2. Release your connection to the database server by using the `CONNECT RESET` statement.
3. Clean up resources used by the program. For example, free any temporary storage or data structures that are used.

Note: If the current transaction is still active when the program terminates, DB2 implicitly ends the transaction. Since DB2's behavior when it implicitly ends a transaction is platform specific, you should explicitly end all transactions by issuing a `COMMIT` or a `ROLLBACK` statement before the program terminates. See *Implicitly Ending a Transaction* for details on how DB2 implicitly ends a transaction.

Implicitly Ending a Transaction

If your program terminates without ending the current transaction, DB2 implicitly ends the current transaction (see “Ending the Program” on page 19 for details on how to properly end your program). DB2 implicitly terminates the current transaction by issuing either a `COMMIT` or a `ROLLBACK` statement when the application ends. Whether DB2 issues a `COMMIT` or `ROLLBACK` depends on factors such as:

- Whether the application terminated normally
- The platform on which the DB2 server runs
- Whether the application uses the context APIs (see “Multiple Thread Database Access” on page 441)

On Most Supported Operating Systems

DB2 implicitly commits a transaction if the termination is normal, or implicitly rolls back the transaction if it is abnormal. Note that what your program considers to be an abnormal termination may not be considered abnormal by the database manager. For example, you may code `exit(-16)` when your application encounters an unexpected error and terminate your application abruptly. The database manager considers this to be a normal termination and commits the transaction. The database manager considers items such as an exception or a segmentation violation as abnormal terminations.

On Windows 32-bit Operating Systems

DB2 always rolls back the transaction regardless of whether your application terminates normally or abnormally, unless you explicitly commit the transaction using the `COMMIT` statement.

When Using the DB2 Context APIs

Your application can use any of the DB2 APIs to set up and pass application contexts between threads as described in “Multiple Thread Database Access” on page 441. If your application uses these DB2 APIs, DB2 implicitly rolls back the transaction regardless of whether your application terminates normally or abnormally. Unless you explicitly commit the transaction using the COMMIT statement, DB2 rolls back the transaction.

Application Pseudocode Framework

Pseudocode Framework for Coding Programs summarizes the general framework for a DB2 application program in pseudocode format. You must, of course, tailor this framework to suit your own program.

Start Program EXEC SQL BEGIN DECLARE SECTION DECLARE USERID FIXED CHARACTER (8) DECLARE PW FIXED CHARACTER (8) (other host variable declarations) EXEC SQL END DECLARE SECTION EXEC SQL INCLUDE SQLCA EXEC SQL WHENEVER SQLERROR GOTO ERRCHK (program logic) EXEC SQL CONNECT TO <i>database A</i> USER :userid USING :pw EXEC SQL SELECT ... EXEC SQL INSERT ... (more SQL statements) EXEC SQL COMMIT (more program logic) EXEC SQL CONNECT TO <i>database B</i> USER :userid USING :pw EXEC SQL SELECT ... EXEC SQL DELETE ... (more SQL statements) EXEC SQL COMMIT (more program logic) EXEC SQL CONNECT TO <i>database A</i> EXEC SQL SELECT ... EXEC SQL DELETE ... (more SQL statements) EXEC SQL COMMIT (more program logic) EXEC SQL CONNECT RESET	<div>Application Setup</div> <div>First Unit of Work</div> <div>Second Unit of Work</div> <div>Third Unit of Work</div>
---	---

Designing an Application For DB2

DB2 provides you with a variety of application development capabilities that you can use to supplement or extend the traditional capabilities of an application. As an application designer, you must make the most fundamental design decision: *Which DB2 capabilities should I use in the design of my application?* In order to make appropriate choices, you need to consider both the database design and target environments for your application. For example, you can choose to enforce some business rules in your database design instead of including the logic in your application.

The capabilities you use and the extent to which you use them can vary greatly. This section is an overview of the capabilities available that can significantly affect your design and provides some reasons for why you might choose one over another. For more information and detail on any of the capabilities described, a reference to more detail is provided.

The capabilities that you need to consider include:

- Accessing the data using:
 - Embedded SQL, including embedded SQLJ for Java (SQLJ)
 - DB2 Call Level Interface (DB2 CLI), Open Database Connectivity (ODBC), and Java Database Connectivity (JDBC)
 - Microsoft Specifications
 - Perl DBI
 - Query products
- Controlling data values using:
 - Data types (built-in or user-defined)
 - Table check constraints
 - Referential integrity constraints
 - Views using the CHECK OPTION
 - Application logic and variable types
- Controlling the relationship between data values using:
 - Referential integrity constraints
 - Triggers
 - Application logic
- Executing programs at the server using:

- Stored procedures
- User-defined functions
- Triggers

You will notice that this list mentions some capabilities more than once, such as triggers. This reflects the flexibility of these capabilities to address more than one design criteria.

Your first and most fundamental decision is whether or not to move the logic to enforce application related rules about the data into the database.

The key advantage in transferring logic focussed on the data from the application into the database is that your application becomes more independent of the data. The logic surrounding your data is centralized in one place, the database. This means that you can change data or data logic once and affect **all** applications immediately.

This latter advantage is very powerful, but you must also consider that any data logic put into the database affects **all** users of the data equally. You must consider whether the rules and constraints that you wish to impose on the data apply to all users of the data or just the users of your application.

Your application requirements may also affect whether to enforce rules at the database or the application. For example, you may need to process validation errors on data entry in a specific order. In general, you should do these types of data validation in the application code.

You should also consider the computing environment where the application is used. You need to consider the difference between performing logic on the client machines against running the logic on the usually more powerful database server machines using either stored procedures, UDFs, or a combination of both.

In some cases, the correct answer is to include the enforcement in both the application (perhaps due to application specific requirements) and in the database (perhaps due to other interactive uses outside the application).

Access to Data

In a relational database, you must use SQL to access the desired data, but you may choose how to integrate the SQL into your application. You can choose from the following interfaces and their supported languages:

Embedded SQL

C/C++, COBOL, FORTRAN, Java (SQLJ), REXX

DB2 CLI and ODBC

C/C++, Java (JDBC)

Microsoft Specifications, including ADO, RDO, and OLE DB

Visual Basic, Visual C++

Perl DBI

Perl

Query Products

Lotus Approach, IBM Query Management Facility

Embedded SQL

Embedded SQL has the advantage that it can consist of either static or dynamic SQL or a mixture of both types. If the content and format of your SQL statements will be *frozen* when your application is in use, you should consider using embedded static SQL in your application. With static SQL, the person who executes the application temporarily inherit the privileges of the user that bound the application to the database. Unless you bind the application with the DYNAMICRULES BIND option, dynamic SQL uses the privileges of the person who executes the application. In general, you should use embedded dynamic SQL where the executable statements are determined at run time. This creates a more secure application program that can handle a greater variety of input.

Note: Embedded SQL for Java (SQLJ) applications can only embed static SQL statements. However, you can use JDBC to make dynamic SQL calls in SQLJ applications.

You must precompile embedded SQL applications to convert the SQL statements into host language commands before using your programming language compiler. In addition, you must bind the SQL in the application to the database for the application to run.

For additional information on using embedded SQL, refer to “Chapter 4. Writing Static SQL Programs” on page 59.

REXX Considerations: REXX applications use APIs which enable them to use most of the features provided by database manager APIs and SQL. Unlike applications written in a compiled language, REXX applications are not precompiled. Instead, a dynamic SQL handler processes all SQL statements. By combining REXX with these callable APIs, you have access to most of the database manager capabilities. Although REXX does not directly support some APIs using embedded SQL, they can be accessed using the DB2 Command Line Processor from within the REXX application.

As REXX is an interpretive language, you may find it is easier to develop and debug your application prototypes in REXX as compared to compiled host languages. Note that while DB2 applications coded in REXX do not provide the performance of DB2 applications that use compiled languages, they do provide the ability to create DB2 applications without precompiling, compiling, linking, or using additional software.

For details of coding and building DB2 applications using REXX, see “Chapter 20. Programming in REXX” on page 607.

DB2 Call Level Interface (DB2 CLI) and Open Database Connectivity (ODBC)

The DB2 Call Level Interface (DB2 CLI) is IBM’s callable SQL interface to the DB2 family of database servers. It is a C and C++ application programming interface for relational database access, and it uses function calls to pass dynamic SQL statements as function arguments. A callable SQL interface is an application program interface (API) for database access, which uses function calls to invoke dynamic SQL statements. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require precompiling or binding.

DB2 CLI is based on the Microsoft™ Open Database Connectivity (ODBC) specification, and the X/Open® specifications. IBM chose these specifications to follow industry standards, and to provide a shorter learning curve for DB2 application programmers who are familiar with either of these database interfaces.

For more information on the ODBC support in DB2, see the *CLI Guide and Reference*.

JDBC: DB2’s Java support includes JDBC, a vendor-neutral dynamic SQL interface that provides data access to your application through standardized Java methods. JDBC is similar to DB2 CLI in that you do not have to precompile or bind a JDBC program. As a vendor-neutral standard, JDBC applications offer increased portability.

An application written using JDBC uses only dynamic SQL. The JDBC interface imposes additional processing overhead.

For additional information on JDBC, refer to “JDBC Programming” on page 540.

Microsoft Specifications

You can write database applications that conform to the ActiveX Data Object (ADO) in Microsoft Visual Basic™ or Visual C++™. ADO applications use the OLE DB Bridge. You can write database applications that conform to the Remote Data Object (RDO) specifications in Visual Basic. You can also define OLE DB table functions that return data from OLE DB providers. For more information on OLE DB table functions, see “OLE DB Table Functions” on page 324.

This book does not attempt to provide a tutorial on writing applications that conform to the ADO and RDO specifications. For full samples of DB2 applications that use the ADO and RDO specifications, refer to the following directories:

- For samples written in Visual Basic, refer to `sqllib\samples\VB`
- For samples written in Visual C++, refer to `sqllib\samples\VC`
- For samples that use the RDO specification, refer to `sqllib\samples\RDO`
- For samples that use the Microsoft Transaction Server™, refer to `sqllib\samples\MTS`

Perl DBI

DB2 supports the Perl Database Interface (DBI) specification for data access through the `DBD::DB2` driver. The `DBD::DB2` driver is supported on the following platforms:

AIX

- Operating Systems
 - Version 4.1.4 and later
- C Compilers
 - IBM C for AIX Version 3.1 and later

HP-UX

- Operating Systems
 - HP-UX Version 10.10 with Patch Levels: PHCO_6134, PHKL_5837, PHKL_6133, PHKL_6189, PHKL_6273, PHSS_5956
 - HP-UX Version 10.20
 - HP-UX Version 11
- C Compilers
 - HP C/HP-UX Version A.10.32
 - HP C Compiler Version A.11.00.00 (for HP-UX Version 11)

Linux

- Operating Systems

- Linux Redhat Version 5.1 with kernel 2.0.35 and glibc version 2.0.7
- C Compilers
 - gcc version 2.7.2.3 or later

Solaris

- Operating Systems
 - Solaris Version 2.5.1
 - Solaris Version 2.6
- C Compilers
 - SPARCompiler C Version 4.2

Windows NT

- Operating Systems
 - Microsoft Windows NT version 4 or later
- C Compilers
 - Microsoft Visual C++ Version 5.0 or later

The DB2 Universal Database Perl DBI website at <http://www.software.ibm.com/data/db2/perl/> contains the latest DBD::DB2 driver and further information.

Query Products

Query products including IBM Query Management Facility (QMF) and Lotus Notes support query development and reporting. The products vary in how SQL statements are developed and the degree of logic that can be introduced. Depending on your needs, this approach may meet your requirements to access data. This book does not provide further information on query products.

Data Value Control

One traditional area of application logic is validating and protecting data integrity by controlling the values allowed in the database. Applications have logic that specifically checks data values as they are entered for validity. (For example, checking that the department number is a valid number and that it refers to an existing department.) There are several different ways of providing these same capabilities in DB2, but from within the database.

Data Types

The database stores every data element in a column of a table, and defines each column with a data type. This data type places certain limits on the types of values for the column. For example, an integer must be a number

within a fixed range. The use of the column in SQL statements must conform to certain behaviors; for instance, the database does not compare an integer to a character string. DB2 includes a set of built-in data types with defined characteristics and behaviors. DB2 also supports defining your own data types, called *user-defined distinct types*, that are based on the built-in types but do not automatically support all the behaviors of the built-in type. You can also use data types, like binary large object (BLOB), to store data that may consist of a set of related values, such as a data structure.

For additional information on data types, refer to the *SQL Reference*.

Unique Constraints

Unique constraints prevent occurrences of duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, you can define a unique constraint on the DEPTNO column in the DEPARTMENT table to ensure that the same department number is not given to two departments.

Use unique constraints if you need to enforce a uniqueness rule for all applications that use the data in a table. For additional information on unique constraints, refer to the *SQL Reference*.

Table Check Constraints

You can use a table check constraint to define restrictions, beyond those of the data type, on the values that are allowed for a column in the table. Table check constraints take the form of range checks or checks against other values in the same row of the same table.

If the rule applies for all applications that use the data, use a table check constraint to enforce your restriction on the data allowed in the table. Table check constraints make the restriction generally applicable and easier to maintain.

For additional information on table check constraints, refer to the *SQL Reference*.

Referential Integrity Constraints

Use referential integrity (RI) constraints if you must maintain value-based relationships for all applications that use the data. For example, you can use an RI constraint to ensure that the value of a DEPTNO column in an EMPLOYEE table matches a value in the DEPARTMENT table. This constraint prevents inserts, updates or deletes that would otherwise result in missing

DEPARTMENT information. By centralizing your rules in the database, RI constraints make the rules generally applicable and easier to maintain.

See “Data Relationship Control” for further uses of RI constraints.

For additional information on referential integrity, refer to the *SQL Reference*.

Views with Check Option

If your application cannot define the desired rules as table check constraints, or the rules do not apply to all uses of the data, there is another alternative to placing the rules in the application logic. You can consider creating a view of the table with the conditions on the data as part of the WHERE clause and the WITH CHECK OPTION clause specified. This view definition restricts the retrieval of data to the set that is valid for your application. Additionally, if you can update the view, the WITH CHECK OPTION clause restricts updates, inserts, and deletes to the rows applicable to your application.

For additional information on the WITH CHECK OPTION, refer to the *SQL Reference*.

Application Logic and Program Variable Types

When you write your application logic in a programming language, you also declare variables to provide some of the same restrictions on data that are described above. In addition, you can choose to write code to enforce rules in the application instead of the database. Place the logic in the application server when:

- The rules are not generally applicable, except in the case of views noted in “Views with Check Option”
- You do not have control over the definitions of the data in the database
- You believe the rule can be more effectively handled in the application logic

For example, processing errors on input data in the order that they are entered may be required, but cannot be guaranteed from the order of operations within the database.

Data Relationship Control

Another major area of focus in application logic is in the area of managing the relationships between different logical entities in your system. For example, if you add a new department, then you need to create a new account code. DB2 provides two methods of managing the relationships between different objects in your database: referential integrity constraints and triggers.

Referential Integrity Constraints

Referential integrity (RI) constraints, considered from the perspective of data relationship control, allow you to control the relationships between data in more than one table. Use the CREATE TABLE or ALTER TABLE statements to define the behavior of operations that affect the related primary key, such as DELETE and UPDATE.

RI constraints enforce your rules on the data across one or more tables. If the rules apply for all applications that use the data, then RI constraints centralize the rules in the database. This makes the rules generally applicable and easier to maintain.

For additional information on referential integrity, refer to the *SQL Reference*.

Triggers

You can use triggers before or after an update to support logic that can also be performed in an application. If the rules or operations supported by the triggers apply for all applications that use the data, then triggers centralize the rules or operations in the database, making it generally applicable and easier to maintain.

For additional information on triggers, see “Chapter 12. Using Triggers in an Active DBMS” on page 373 and refer to the *SQL Reference*.

Using Triggers Before an Update: Using triggers that run before an update or insert, values that are being updated or inserted can be modified before the database is actually modified. These can be used to transform input from the application (user view of the data) to an internal database format where desired. These *before triggers* can also be used to cause other non-database operations to be activated through user-defined functions.

Using Triggers After an Update: Triggers that run after an update, insert or delete can be used in several ways:

- Triggers can update, insert, or delete data in the same or other tables. This is useful to maintain relationships between data or to keep audit trail information.
- Triggers can check data against values of data in the rest of the table or in other tables. This is useful when you cannot use RI constraints or check constraints because of references to data from other rows from this or other tables.
- Triggers can use user-defined functions to activate non-database operations. This is useful, for example, for issuing alerts or updating information outside the database.

Application Logic

You may decide to write code to enforce rules or perform related operations in the application instead of the database. You must do this for cases where you cannot generally apply the rules to the database. You may also choose to place the logic in the application when you do not have control over the definitions of the data in the database or you believe the application logic can handle the rules or operations more efficiently.

Application Logic at the Server

A final aspect of application design for which DB2 offers additional capability is running some of your application logic at the database server. Usually you will choose this design to improve performance, but you may also run application logic at the server to support common functions.

Stored Procedures

A stored procedure is a routine for your application that is called from client application logic but runs on the database server. The most common reason to use a stored procedure is for database intensive processing that produces only small amounts of result data. This can save a large amount of communications across the network during the execution of the stored procedure. You may also consider using a stored procedure for a set of operations that are common to multiple applications. In this way, all the applications use the same logic to perform the operation.

For additional information on Stored Procedures, refer to “Chapter 6. Stored Procedures” on page 185.

User-Defined Functions

You can write a user-defined function (UDF) for use in performing operations within an SQL statement to return:

- A single scalar value (scalar function)
- A table from a non-DB2 data source, for example, an ASCII file or a Web page (table function)

A UDF cannot contain SQL statements. UDFs are useful for tasks like transforming data values, performing calculations on one or more data values, or extracting parts of a value (such as extracting parts of a large object).

For additional information on writing user-defined functions, refer to “Chapter 11. Writing User-Defined Functions (UDFs)” on page 287.

Triggers

In “Triggers” on page 30, it is noted that triggers can be used to invoke user-defined functions. This is useful when you always want a certain non-SQL operation performed when specific statements occur, or data values are changed. Examples include such operations as issuing an electronic mail message under specific circumstances or writing alert type information to a file.

For additional information on triggers, refer to “Chapter 12. Using Triggers in an Active DBMS” on page 373.

Supported SQL Statements

The SQL language provides for data definition, retrieval, update, and control operations from within an application. Table 26 on page 629 shows the SQL statements supported by the DB2 product and whether the statement is supported dynamically, through the CLP, or through the DB2 CLI. You can use Table 26 on page 629 as a quick reference aid. For a complete discussion of all the statements, including their syntax, refer to the *SQL Reference*.

Authorization Considerations

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way. DB2 uses a set of privileges to provide protection for the information that you store in it. For more information about the different privileges, refer to the *Administration Guide, Design and Implementation*.

Most SQL statements require some type of privilege on the database objects which the statement utilizes. Most API calls usually do not require any privilege on the database objects which the call utilizes, however, many APIs require that you possess the necessary authority in order to invoke them. The DB2 APIs enable you to perform the DB2 administrative functions from within your application program. For example, to recreate a package stored in the database without the need for a bind file, you can use the `sqlarbnd` (or `REBIND`) API. For details on each DB2 API, refer to the *Administrative API Reference*.

For information on the required privilege to issue each SQL statement, refer to the *SQL Reference*. For information on the required privilege and authority to issue each API call, refer to the *Administrative API Reference*.

When you design your application, consider the privileges your users will need to run the application. The privileges required by your users depend on:

- whether your application uses dynamic SQL, including JDBC and DB2 CLI, or static SQL
- which APIs the application uses

Dynamic SQL

To use dynamic SQL in a package bound with DYNAMICRULES RUN (default), the person that runs a dynamic SQL application must have the privileges necessary to issue each SQL request performed, as well as the EXECUTE privilege on the package. The privileges may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC.

If you bind the application with the DYNAMICRULES BIND option, DB2 associates your authorization ID with the application packages. This allows any user that runs the application to inherit the privileges associated your authorization ID.

The person binding the application (for embedded dynamic SQL applications) only needs the BINDADD authority on the database, if the program contains no static SQL. Again, this privilege can be granted to the user's authorization ID, to a group of which the user is a member, or to PUBLIC.

When you bind a dynamic SQL package with the DYNAMICRULES BIND option, the user that runs the application only needs the EXECUTE privilege on the package. To bind a dynamic SQL application with the DYNAMICRULES BIND option, you must have the privileges necessary to perform all the dynamic and static SQL statements in the application. If you have SYSADM or DBADM authority and bind packages with DYNAMICRULES BIND, consider using the OWNER BIND option to designate a different authorization ID. OWNER BIND prevents the package from automatically inheriting SYSADM or DBADM privileges on dynamic SQL statements. For more information on DYNAMICRULES BIND and OWNER BIND, refer to the BIND command in the *Command Reference*.

Static SQL

To use static SQL, the user running the application only needs the EXECUTE privilege on the package. No privileges are required for each of the statements that make up the package. The EXECUTE privilege may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC.

The person binding the application, however, must have the privileges necessary to perform all the statements in the application, in addition to the BINDADD authority. The privileges needed to execute the statements must be granted to the user's authorization ID or to PUBLIC. Group privileges are not used when binding static SQL statements. As with dynamic SQL, the BINDADD privilege can be granted to the user authorization ID, to a group of which the user is a member, or to PUBLIC.

These properties of static SQL give you very precise control over access to information in DB2. See the example at the end of this section for a possible application of this.

Using APIs

Most of the APIs provided by DB2 do not require the use of privileges, however, many do require some kind of authority to invoke. For the APIs that do require a privilege, the privilege must be granted to the user running the application. The privilege may be granted to the user's authorization ID, to any group of which the user is a member, or to PUBLIC. For information on the required privilege and authority to issue each API call, see the *Administrative API Reference*.

Example

Consider two users, PAYROLL and BUDGET, who need to perform queries against the STAFF table. PAYROLL is responsible for paying the employees of the company, so it needs to issue a variety of SELECT statements when issuing paychecks. PAYROLL needs to be able to access each employee's salary. BUDGET is responsible for determining how much money is needed to pay the salaries. BUDGET should not, however, be able to see any particular employee's salary.

Since PAYROLL issues many different SELECT statements, the application you design for PAYROLL could probably make good use of dynamic SQL. This would require that PAYROLL have SELECT privilege on the STAFF table. This is not a problem since PAYROLL needs full access to the table anyhow.

BUDGET, on the other hand, should not have access to each employee's salary. This means that you should not grant SELECT privilege on the STAFF table to BUDGET. Since BUDGET does need access to the total of all the salaries in the STAFF table, you could build a static SQL application to execute a SELECT SUM(SALARY) FROM STAFF, bind the application and grant the EXECUTE privilege on your application's package to BUDGET. This lets BUDGET get the needed information without exposing the information that BUDGET should not see.

Database Manager APIs Used in Embedded SQL or DB2 CLI Programs

Your application can use APIs to access database manager facilities that are not available using SQL statements. For complete details on the APIs available with the database manager and how to call them, refer to the examples in the *Administrative API Reference*.

You can use the DB2 APIs to:

- Manipulate the database manager environment, which includes cataloging and uncataloging of databases and nodes, and scanning database and node directories. You can also use APIs to create, delete, and migrate databases
- Provide facilities to import and export data, and administer, backup, and restore the database
- Manipulate the database manager configuration file and the database configuration files
- Provide operations specific to the client/server environment
- Provide the run-time interface for precompiled SQL statements. These APIs are not usually called directly by the programmer. Instead, they are inserted into the modified source file by the precompiler after processing.

The database manager includes APIs for language vendors who want to write their own precompiler, and other APIs useful for developing applications.

For complete details on the APIs available with the database manager and how to call them, see the examples in the *Administrative API Reference*.

Setting Up the Testing Environment

In order to perform many of the tasks described in the following sections, you should set up a test environment. For example, you need a database to test your application's SQL code.

A testing environment should include the following:

- **A test database.** If your application updates, inserts, or deletes data from tables and views, use test data to verify its execution. If it only retrieves data from tables and views, consider using production-level data when testing it.
- **Test input data.** The input data used to test an application should be valid data that represents all possible input conditions. If the application verifies that input data is valid, include both valid and invalid data to verify that the valid data is processed and the invalid data is flagged.

Creating a Test Database

If you must create a test database, write a small server application that calls the CREATE DATABASE API, or use the command line processor. Refer to the *Command Reference* for information about the command line processor, or the *Administrative API Reference* for information about the CREATE DATABASE API.

Creating Test Tables

To design the test tables and views needed, first analyze the data needs of the application. To create a table, you need the CREATETAB authority and the CREATEIN privilege on the schema. Refer to the information on the CREATE TABLE statement in the *SQL Reference* for alternative authorities.

List the data the application accesses and describe how each data item is accessed. For example, suppose the application being developed accesses the TEST.TEMPL, TEST.TDEPT, and TEST.TPROJ tables. You could record the type of accesses as shown in Table 1.

Table 1. Description of the Application Data

Table or View Name	Insert Rows	Delete Rows	Column Name	Data Type	Update Access
TEST.TEMPL	No	No	EMPNO	CHAR(6)	Yes
			LASTNAME	VARCHAR(15)	Yes
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	
			JOBCODE	DECIMAL(3)	
TEST.TDEPT	No	No	DEPTNO	CHAR(3)	
			MGRNO	CHAR(6)	
TEST.TPROJ	Yes	Yes	PROJNO	CHAR(6)	Yes
			DEPTNO	CHAR(3)	Yes
			RESPEMP	CHAR(6)	Yes
			PRSTAFF	DECIMAL(5,2)	Yes
			PRSTDATE	DECIMAL(6)	Yes
			PRENDATE	DECIMAL(6)	

When the description of the application data access is complete, construct the test tables and views that are needed to test the application:

- Create a test table when the application modifies data in a table or a view. Create the following test tables using the CREATE TABLE SQL statement:
 - TEMPL
 - TPROJ
- Create a test view when the application does not modify data in the production database.

In this example, create a test view of the TDEPT table using the CREATE VIEW SQL statement.

If the database schema is being developed along with the application, the definitions of the test tables might be refined repeatedly during the development process. Usually, the primary application cannot both create the tables and access them because the database manager cannot bind statements that refer to tables and views that do not exist. To make the process of creating and changing tables less time-consuming, consider developing a separate application to create the tables. Of course you can always create test tables interactively using the Command Line Processor (CLP).

Generating Test Data

Use any of the following methods to insert data into a table:

- INSERT...VALUES (an SQL statement) puts one or more rows into a table each time the command is issued.
- INSERT...SELECT obtains data from an existing table (based on a SELECT clause) and puts it into the table identified with the INSERT statement.
- The IMPORT or LOAD utility inserts large amounts of new or existing data from a defined source.
- The RESTORE utility can be used to duplicate the contents of an existing database into an identical test database by using a BACKUP copy of the original database.

For information about the INSERT statement, refer to the *SQL Reference*. For information about the IMPORT, LOAD, and RESTORE utilities, refer to the *Administration Guide*.

The following SQL statements demonstrate a technique you can use to populate your tables with randomly generated test data. Suppose the table EMP contains four columns, ENO (employee number), LASTNAME (last name), HIREDATE (date of hire) and SALARY (employee's salary) as in the following CREATE TABLE statement:

```
CREATE TABLE EMP (ENO INTEGER, LASTNAME VARCHAR(30),  
                   HIREDATE DATE, SALARY INTEGER);
```

Suppose you want to populate this table with employee numbers from 1 to a number, say 100, with random data for the rest of the columns. You can do this using the following SQL statement:

```
INSERT INTO EMP  
-- generate 100 records  
WITH DT(ENO) AS (VALUES(1) UNION ALL  
SELECT ENO+1 FROM DT WHERE ENO < 100 )
```

```
-- Now, use the generated records in DT to create other columns
```

```
-- of the employee record.
SELECT ENO, 2
      TRANSLATE(CHAR(INTEGER(RAND()*1000000)), 3
        CASE MOD(ENO,4) WHEN 0 THEN 'aeiou' | 'bcdfg'
                      WHEN 1 THEN 'aeiou' | 'hklm'
                      WHEN 2 THEN 'aeiou' | 'npqrs'
                      ELSE 'aeiou' | 'twxyz' END,
        '1234567890') AS LASTNAME,
      CURRENT DATE - (RAND()*10957) DAYS AS HIREDATE, 4
      INTEGER(10000+RAND()*200000) AS SALARY 5
FROM DT;

SELECT * FROM EMP;
```

The following is an explanation of the above statement:

1. The first part of the INSERT statement generates 100 records for the first 100 employees using a recursive subquery to generate the employee numbers. Each record contains the employee number. To change the number of employees, use a number other than 100.
2. The SELECT statement generates the LASTNAME column. It begins by generating a random integer up to 6 digits long using the RAND function. It then converts the integer to its numeric character format using the CHAR function.
3. To convert the numeric characters to alphabet characters, the statement uses the TRANSLATE function to convert the ten numeric characters (0 through 9) to alphabet characters. Since there are more than 10 alphabet characters, the statement selects from five different translations. This results in names having enough random vowels to be pronounceable and so the vowels are included in each translation.
4. The statement generates a random HIREDATE value. The value of HIREDATE ranges back from the current date to 30 years ago. HIREDATE is calculated by subtracting a random number of days between 0 and 10 957 from the current date. (10 957 is the number of days in 30 years.)
5. Finally, the statement randomly generates the SALARY. The minimum salary is 10 000, to which a random number from 0 to 200 000 is added.

For sample programs that are helpful in generating random test data, please see the fillcli.sqc and fillsrv.sqc sample programs in the sqllib/samples/c subdirectory.

You may also want to consider prototyping any user-defined functions (UDF) you are developing against the test data. For more information on why and how you write UDFs, see “Chapter 11. Writing User-Defined Functions (UDFs)” on page 287 and “Chapter 9. User-Defined Functions (UDFs)” on page 255.

Running, Testing and Debugging Your Programs

The *Application Building Guide* tells you how to run your program in your environment. You can do the following to help you during the testing and debugging of your code:

- Use the same techniques discussed in “Prototyping Your SQL Statements”. These include using the command line processor, the Explain facility, analyzing the system catalog views for information about the tables and databases your program is manipulating, and updating certain system catalog statistics to simulate production conditions.
- Use the database system monitor to capture certain optimizing information for analysis. See the *System Monitor Guide and Reference*.
- Use the flagger facility to check the syntax of SQL statements in applications being developed for DB2 Universal Database for OS/390, or for conformance to the SQL92 Entry Level standard. This facility is invoked during precompilation. For information about how to do this, see “Precompiling” on page 47, towards the end of the section.
- Make full use of the error-handling APIs. For example, you can use error-handling APIs to print all messages during the testing phase. For more information about error-handling APIs, see the *Administrative API Reference*.

Prototyping Your SQL Statements

As you design and code your application, you can take advantage of certain database manager features and utilities to prototype portions of your SQL code, and to improve performance. For example, you can do the following:

- Use the command line processor (CLP) to test many SQL statements before you attempt to compile and link a complete program.

This allows you to define and manipulate information stored in a database table, index, or view. You can add, delete, or update information as well as generate reports from the contents of tables. Note that you have to minimally change the CLP syntax for some SQL statements in order to use host variables in your embedded SQL program. Host variables are used to store data that is output to your screen using the CLP. In addition, some embedded SQL statements (such as BEGIN DECLARE SECTION) are not supported by the CLP as they are not relevant to that environment. See Table 26 on page 629 to see which SQL statements are not supported by the CLP.

You can also redirect the input and output of command line processor requests. For example, you could create one or more files containing SQL statements you need as input into a command line processor request, to save retyping the statement.

For information about the command line processor, refer to the *Command Reference*.

- Use the Explain facility to get an idea of the estimated costs of the DELETE, INSERT, UPDATE, or SELECT statements you plan to use in your program. The Explain facility places the information about the structure and the estimated costs of the subject statement into user supplied tables. You can view this information using Visual Explain or the db2exfmt utility.

For information about how to use the Explain facility, refer to the *Administration Guide, Performance*.

- Use the system catalog views to easily retrieve information about existing databases. The database manager creates and maintains the system catalog tables on which the views are based during normal operation as databases are created, altered, and updated. These views contain data about each database, including authorities granted, column names, data types, indexes, package dependencies, referential constraints, table names, views, and so on. Data in the system catalog views is available through normal SQL query facilities.

You can update some system catalog views containing statistical information used by the SQL optimizer. You may change some columns in these views to influence the optimizer or to investigate the performance of hypothetical databases. You can use this method to simulate a production system on your development or test system and analyze how queries perform.

For a complete description of each system catalog view, refer to the appendix in the *SQL Reference*. For information about system catalog statistics and which ones you can change, refer to the *Administration Guide, Performance*.

Part 2. Embedding SQL in Applications

Chapter 3. Embedded SQL Overview

Embedding SQL Statements in a Host Language	43	Resolving Unqualified Table Names	52
Creating and Preparing the Source Files	45	Other Binding Considerations	53
Creating Packages for Embedded SQL	47	Advantages of Deferred Binding	54
Precompiling	47	DB2 Bind File Description Utility - db2bfd	54
Source File Requirements	49	Application, Bind File, and Package Relationships	55
Compiling and Linking	50	Timestamps	56
Binding	51	Rebinding	57
Renaming Packages	51		
Binding Dynamic Statements.	52		

Embedding SQL Statements in a Host Language

You can write applications with SQL statements embedded within a host language. The SQL statements provide the database interface, while the host language provides the remaining support needed for the application to execute.

Table 2 shows an SQL statement embedded in a host language application. In the example, the application checks the SQLCODE field of the SQLCA structure to determine whether the update was successful.

Table 2. Embedding SQL Statements in a Host Language

Language	Sample Source Code
C/C++	<pre>EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'; if (SQLCODE < 0) printf("Update Error: SQLCODE = %ld \n", SQLCODE);</pre>
Java (SQLJ)	<pre>try { #sql { UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' }; } catch (SQLException e) { println("Update Error: SQLCODE = " + e.getErrorCode()); }</pre>
COBOL	<pre>EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC. IF SQLCODE LESS THAN 0 DISPLAY 'UPDATE ERROR: SQLCODE = ', SQLCODE.</pre>
FORTRAN	<pre>EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' if (sqlcode .lt. 0) THEN write(*,*) 'Update error: sqlcode = ', sqlcode</pre>

SQL statements placed in an application are not specific to the host language. The database manager provides a way to convert the SQL syntax for processing by the host language.

For the C, C++, COBOL or FORTRAN languages, this conversion is handled by the DB2 precompiler. The DB2 precompiler is invoked using the PREP command. The precompiler converts embedded SQL statements directly into DB2 run-time services API calls.

For the Java language, the SQLJ translator converts SQLJ clauses into JDBC statements. The SQLJ translator is invoked with the SQLJ command.

When the precompiler processes a source file, it specifically looks for SQL statements and avoids the non-SQL host language. It can find SQL statements because they are surrounded by special delimiters. For the syntax information necessary to embed SQL statements in the language you are using, see the following:

- for C/C++, “Embedding SQL Statements” on page 497
- for Java (SQLJ), “Embedding SQL Statements” on page 548
- for COBOL, “Embedding SQL Statements” on page 573
- for FORTRAN, “Embedding SQL Statements” on page 595
- for REXX, “Embedding SQL Statements” on page 609

Table 3 shows how to use delimiters and comments to create valid embedded SQL statements in the supported compiled host languages.

Table 3. Embedding SQL Statements in a Host Language

Language	Sample Source Code
C/C++	<pre>/* Only C or C++ comments allowed here */ EXEC SQL -- SQL comments or /* C comments or */ // C++ comments allowed here DECLARE C1 CURSOR FOR sname; /* Only C or C++ comments allowed here */</pre>
SQLJ	<pre>/* Only Java comments allowed here */ #sql c1 = { -- SQL comments or /* Java comments or */ // Java comments allowed here SELECT name FROM employee }; /* Only Java comments allowed here */</pre>

Table 3. Embedding SQL Statements in a Host Language (continued)

Language	Sample Source Code
COBOL	<ul style="list-style-type: none"> * See COBOL documentation for comment rules * Only COBOL comments are allowed here EXEC SQL -- SQL comments or * full-line COBOL comments are allowed here DECLARE C1 CURSOR FOR sname END-EXEC. * Only COBOL comments are allowed here
FORTRAN	<pre> C Only FORTRAN comments are allowed here EXEC SQL + -- SQL comments, and C full-line FORTRAN comment are allowed here + DECLARE C1 CURSOR FOR sname I=7 ! End of line FORTRAN comments allowed here C Only FORTRAN comments are allowed here </pre>

Creating and Preparing the Source Files

You can create the source code in a standard ASCII file, called a source file, using a text editor. The source file must have the proper extension for the host language in which you write your code. See Table 27 on page 635 to find out the required file extension for the host language you are using.

Note: Not all platforms support all host languages. See the *Application Building Guide* for specific information.

For this discussion, assume that you have already written the source code.

If you have written your application using a compiled host language, you must follow additional steps to build your application. Along with compiling and linking your program, you must *precompile* and *bind* it.

Simply stated, precompiling converts embedded SQL statements into DB2 run-time API calls that a host compiler can process, and creates a bind file. The bind file contains information on the SQL statements in the application program. The BIND command creates a *package* in the database. Optionally, the precompiler can perform the bind step at precompile time.

Binding is the process of creating a *package* from a bind file and storing it in a database. If your application accesses more than one database, you must create a package for each database.

Figure 1 on page 46 shows the order of these steps, along with the various modules of a typical compiled DB2 application. You may wish to refer to it as

you read through the following sections about what happens at each stage of program preparation.

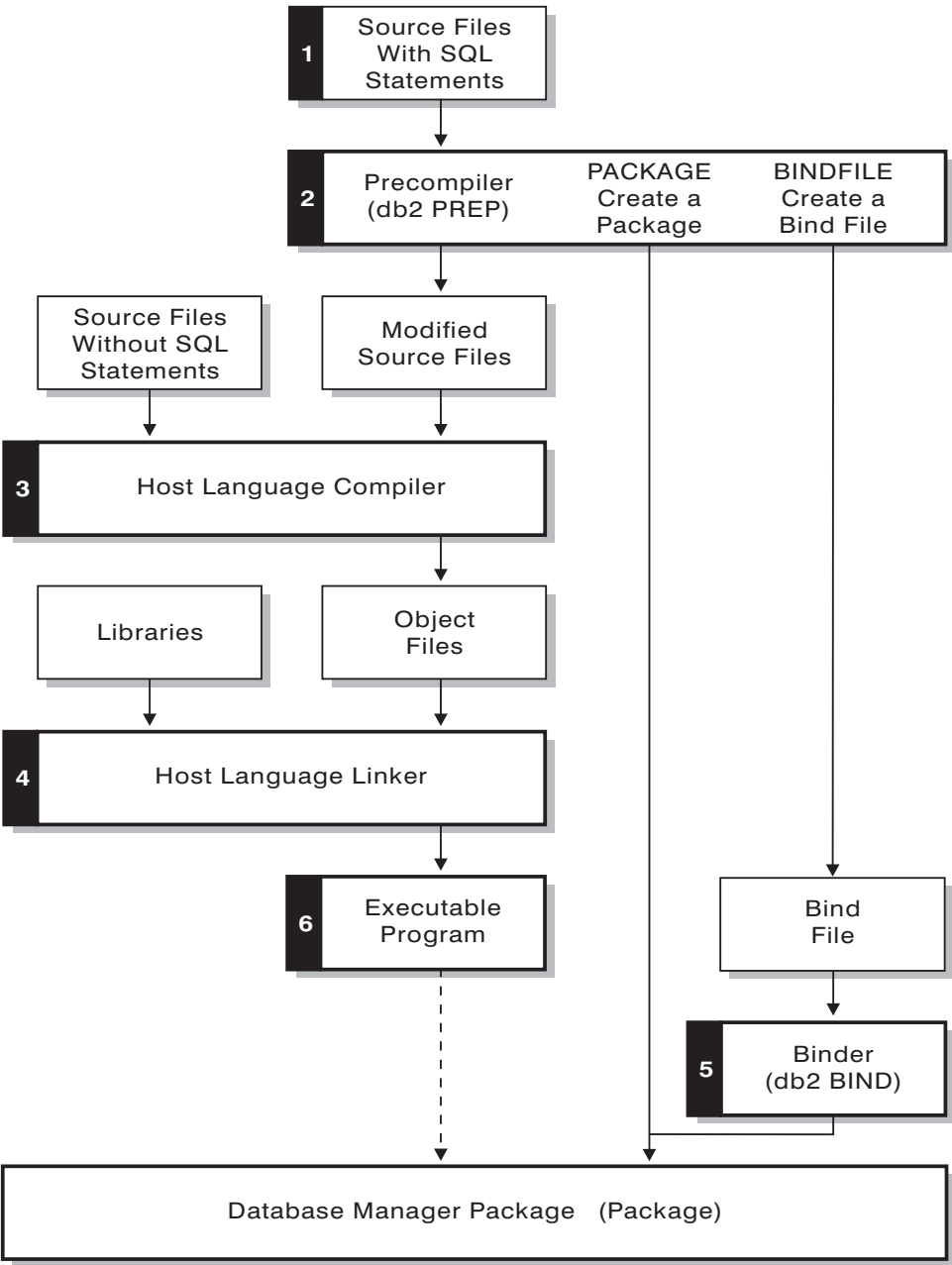


Figure 1. Preparing Programs Written in Compiled Host Languages

Creating Packages for Embedded SQL

To run applications written in compiled host languages, you must create the packages needed by the database manager at execution time. This involves the following steps as shown in Figure 1 on page 46:

- Precompiling (step 2), to convert embedded SQL source statements into a form the database manager can use,
- Compiling and Linking (steps 3 and 4), to create the required object modules, and,
- Binding (step 5), to create the package to be used by the database manager when the program is run.

Other topics discussed in this section include:

- Application, Bind File, and Package Relationships, and,
- Rebinding, which describes when and how to rebind packages.

To create the packages needed by SQLJ applications, you need to use both the SQLJ translator and db2prof command. For more information on using the SQLJ translator, see “SQLJ Programming” on page 545.

Precompiling

After you create the source files, you must precompile each host language file containing SQL statements with the PREP command for host language source files. The precompiler converts SQL statements contained in the source file to comments, and generates the DB2 run-time API calls for those statements.

Before precompiling an application you must connect to a server, either implicitly or explicitly. Although you precompile application programs at the client workstation and the precompiler generates modified source and messages on the client, the precompiler uses the server connection to perform some of the validation.

The precompiler also creates the information the database manager needs to process the SQL statements against a database. This information is stored in a package, in a bind file, or in both, depending on the precompiler options selected.

A typical example of using the precompiler follows. To precompile a C embedded SQL source file called *filename.sqc*, you can issue the following command to create a C source file with the default name *filename.c* and a bindfile with the default name *filename.bnd*:

```
DB2 PREP filename.sqc BINDFILE
```

For detailed information on precompiler syntax and options, see the *Command Reference*.

The precompiler generates up to four types of output:

- Modified source
- Package
- Bind file
- Message file

Modified Source

This file is the new version of the original source file after the precompiler converts the SQL statements into DB2 run-time API calls. It is given the appropriate host language extension.

Package

If you use the PACKAGE option (the default), or do not specify any of the BINDFILE, SYNTAX, or SQLFLAG options, the package is stored in the connected database. The package contains all the information required to execute the static SQL statements of a particular source file against this database only. Unless you specify a different name with the PACKAGE USING option, the precompiler forms the package name from the first 8 characters of the source file name.

With the PACKAGE option, the database used during the precompile process must contain all of the database objects referenced by the static SQL statements in the source file. For example, you cannot precompile a SELECT statement unless the table it references exists in the database.

Bind File

If you use the BINDFILE option, the precompiler creates a bind file (with extension .bnd) that contains the data required to create a package. This file can be used later with the BIND command to bind the application to one or more databases. If you specify BINDFILE and do not specify the PACKAGE option, binding is deferred until you invoke the BIND command. Note that for the Command Line Processor (CLP), the default for PREP does not specify the BINDFILE option. Thus, if you are using the CLP and want the binding to be deferred, you need to specify the BINDFILE option.

If you request a bind file at precompile time but do not specify the PACKAGE, that is, you do not create a package, certain object existence and authorization SQLCODEs are treated as warnings instead of errors. This enables you to precompile a program and create a bind file without requiring that the referenced objects be present, or requiring that you possess the authority to execute the SQL statements being

precompiled. For a list of the specific SQLCODEs that are treated as warnings instead of errors refer to the *Command Reference*.

Message File If you use the MESSAGES option, the precompiler redirects messages to the indicated file. These messages include warnings and error messages that describe problems encountered during precompilation. If the source file does not precompile successfully, use the warning and error messages to determine the problem, correct the source file, and then attempt to precompile the source file again. If you do not use the MESSAGES option, precompilation messages are written to the standard output.

Source File Requirements

You must always precompile a source file against a specific database, even if eventually you do not use the database with the application. In practice, you can use a test database for development, and after you fully test the application, you can bind its bind file to one or more production databases. See “Advantages of Deferred Binding” on page 54 for other ways to use this feature.

If your application uses a code page that is not the same as your database code page, you need to consider which code page to use when precompiling. See “Conversion Between Different Code Pages” on page 407.

If your application uses user-defined functions (UDFs) or user-defined distinct types (UDTs), you may need to use the FUNCPATH option when you precompile your application. This option specifies the function path that is used to resolve UDFs and UDTs for applications containing static SQL. If FUNCPATH is not specified, the default function path is *SYSIBM*, *SYSFUN*, *USER*, where *USER* refers to the current user ID. For more information on bind options refer to the *Command Reference*.

To precompile an application program that accesses more than one server, you can do one of the following:

- Split the SQL statements for each database into separate source files. Do not mix SQL statements for different databases in the same file. Each source file can be precompiled against the appropriate database. This is the recommended method.
- Code your application using dynamic SQL statements only, and bind against each database your program will access.
- If all the databases look the same, that is, they have the same definition, you can group the SQL statements together into one source file.

The same procedures apply if your application will access a DRDA application server through DB2 Connect. Precompile it against the server to which it will be connecting, using the PREP options available for that server.

If you are precompiling an application that will run on DB2 Universal Database for OS/390, consider using the flagger facility to check the syntax of the SQL statements. The flagger indicates SQL syntax that is supported by DB2 Universal Database, but not supported by DB2 Universal Database for OS/390. You can also use the flagger to check that your SQL syntax conforms to the SQL92 Entry Level syntax. You can use the SQLFLAG option on the PREP command to invoke it and to specify the version of DB2 Universal Database for OS/390 SQL syntax to be used for comparison. The flagger facility will not enforce any changes in SQL use; it only issues informational and warning messages regarding syntax incompatibilities, and does not terminate preprocessing abnormally.

For details about the PREP command, refer to the *Command Reference*.

Compiling and Linking

Compile the modified source files and any additional source files that do not contain SQL statements using the appropriate host language compiler. The language compiler converts each modified source file into an *object module*.

Refer to the *Application Building Guide* or other programming documentation for your operating platform for any exceptions to the default compile options. Refer to your compiler's documentation for a complete description of available compile options.

The host language linker creates an executable application. For example:

- On OS/2 and Windows 32-bit operating systems, the application can be an executable file or a dynamic link library (DLL).
- On UNIX-based systems, the application can be an executable load module or a shared library.

Note: Although applications can be DLLs on Windows 32-bit operating systems, the DLLs are loaded directly by the application and not by the DB2 database manager. On Windows 32-bit operating systems, the database manager can load DLLs. Stored procedures are normally built as DLLs or shared libraries. For information on using stored procedures, see “Chapter 6. Stored Procedures” on page 185.

For information on creating executable files on other platforms supported by DB2, refer to the *Application Building Guide*.

To create the executable file, link the following:

- User object modules, generated by the language compiler from the modified source files and other files not containing SQL statements
- Host language library APIs, supplied with the language compiler
- The database manager library containing the database manager APIs for your operating environment. Refer to the *Application Building Guide* or other programming documentation for your operating platform for the specific name of the database manager library you need for your database manager APIs.

Binding

Binding is the process that creates the package the database manager needs in order to access the database when the application is executed. Binding can be done implicitly by specifying the PACKAGE option during precompilation, or explicitly by using the BIND command against the bind file created during precompilation.

A typical example of using the BIND command follows. To bind a bind file named *filename.bnd* to the database, you can issue the following command:

```
DB2 BIND filename.bnd
```

For detailed information on BIND command syntax and options, refer to the *Command Reference*.

One package is created for each separately precompiled source code module. If an application has five source files, of which three require precompilation, three packages or bind files are created. By default, each package is given a name that is the same as the name of the source module from which the .bnd file originated, but truncated to 8 characters. If the name of this newly created package is the same as a package that currently exists in the target database, the new package replaces the previously existing package. To explicitly specify a different package name, you must use the PACKAGE USING option on the PREP command. See the *Command Reference* for details.

Renaming Packages

When creating multiple versions of an application, you should avoid conflicting names by renaming your package. For example, if you have an application called foo (compiled from foo.sqc), you precompile it and send it to all the users of your application. The users bind the application to the database, and then run the application. To make subsequent changes, create a new version of foo and send this application and its bind file to the users that require the new version. The new users bind foo.bnd and the new application runs without any problem. However, when users attempt to run the old version of the application, they receive a timestamp conflict on the F00 package (which indicates that the package in the database does not match the

application being run) so they rebind the client. (See “Timestamps” on page 56 for more information on package timestamps.) Now the users of the new application receive a timestamp conflict. This problem is caused because both applications use packages with the same name.

The solution is to use package renaming. When you build the first version of F00, you precompile it with the command:

```
DB2 PREP F00.SQC BINDFILE PACKAGE USING F001
```

After you distribute this application, users can bind and run it without any problem. When you build the new version, you precompile it with the command:

```
DB2 PREP F00.SQC BINDFILE PACKAGE USING F002
```

After you distribute the new application, it will also bind and run without any problem. Since the package name for the new version is F002 and the package name for the first version is F001, there is no naming conflict and both versions of the application can be used.

Binding Dynamic Statements

For dynamically prepared statements, the values of a number of special registers determine the statement compilation environment:

- The CURRENT QUERY OPTIMIZATION special register determines which optimization class is used.
- The CURRENT FUNCTION PATH special register determines the function path used for UDF and UDT resolution.
- The CURRENT EXPLAIN SNAPSHOT register determines whether explain snapshot information is captured.
- The CURRENT EXPLAIN MODE register determines whether explain table information is captured, for any eligible dynamic SQL statement. The default values for these special registers are the same defaults used for the related bind options. For information on special registers and their interaction with BIND options, refer to the appendix of the *SQL Reference*.

Resolving Unqualified Table Names

You can handle unqualified table names in your application by using one of the following methods:

- For each user, bind the package with different COLLECTION parameters from different authorization identifiers by using the following commands:

```
CONNECT TO db_name USER user_name  
BIND file_name COLLECTION schema_name
```


In the above example, *db_name* is the name of the database, *user_name* is the name of the user, and *file_name* is the name of the application that will be bound. Note that *user_name* and *schema_name* are usually the same value. Then use the SET CURRENT PACKAGESET statement to specify which package to use, and therefore, which qualifiers will be used. The default qualifier is the authorization identifier that is used when binding the package. For an example of how to use the SET CURRENT PACKAGESET statement, refer to the *SQL Reference*.

- Create views for each user with the same name as the table so the unqualified table names resolve correctly. (Note that the QUALIFIER option is DB2 Connect only, meaning that it can only be used when using a host server.)
- Create an alias for each user to point to the desired table.

Other Binding Considerations

If your application code page uses a different code page from your database code page, you may need to consider which code page to use when binding. See “Conversion Between Different Code Pages” on page 407.

If your application issues calls to any of the database manager utility APIs, such as IMPORT or EXPORT, you must bind the supplied utility bind files to the database. For details, refer to the *Quick Beginnings* guide for your platform.

You can use bind options to control certain operations that occur during binding, as in the following examples:

- The QUERYOPT bind option takes advantage of a specific optimization class when binding.
- The EXPLSNAP bind option stores Explain Snapshot information for eligible SQL statements in the Explain tables.
- The FUNCPATH bind option properly resolves user-defined distinct types and user-defined functions in static SQL.

For information on bind options, refer to the section on the BIND command in the *Command Reference*.

If the bind process starts but never returns, it may be that other applications connected to the database hold locks that you require. In this case, ensure that no applications are connected to the database. If they are, disconnect all applications on the server and the bind process will continue.

If your application will access a server using DB2 Connect, you can use the BIND options available for that server. For details on the BIND command and its options, refer to the *Command Reference*.

Bind files are not backward compatible with previous versions of DB2 Universal Database. In mixed-level environments, DB2 can only use the functions available to the lowest level of the database environment. For example, if a V5.2 client connects to a V5.0 server, the client will only be able to use V5.0 functions. As bind files express the functionality of the database, they are subject to the mixed-level restriction.

If you need to rebind higher-level bind files on lower-level systems, you can:

- Use a lower-level DB2 client that has the SDK installed to connect to the higher-level server and create bind files which can be shipped and bound to the lower-level DB2 Universal Database environment.
- Use a higher-level DB2 client in the lower-level production environment to bind the higher-level bind files that were created in the test environment. The higher-level client passes only the options that apply to the lower-level server.

Advantages of Deferred Binding

Precompiling with binding enabled allows an application to access only the database used during the precompile process. Precompiling with binding deferred, however, allows an application to access many databases, because you can bind the BIND file against each one. This method of application development is inherently more flexible in that applications are precompiled only once, but the application can be bound to a database at any time.

Using the BIND API during execution allows an application to bind itself, perhaps as part of an installation procedure or before an associated module is executed. For example, an application can perform several tasks, only one of which requires the use of SQL statements. You can design the application to bind itself to a database only when the application calls the task requiring SQL statements, and only if an associated package does not already exist.

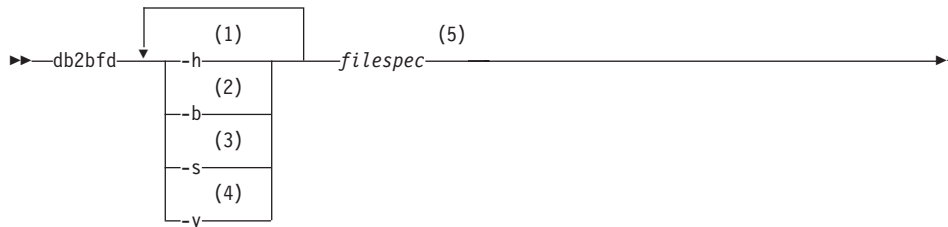
Another advantage of the deferred binding method is that it lets you create packages without providing source code to end users. You can ship the associated bind files with the application.

DB2 Bind File Description Utility - db2bfd

With the DB2 Bind File Description (db2bfd) utility, you can easily display the contents of a bind file to examine and verify the SQL statements within it, as well as display the precompile options used to create the bind file. This may be useful in problem determination related to your application's bind file.

The db2bfd utility is located in the bin subdirectory of the sqllib directory of the instance.

Its syntax is:



Notes:

1. Display the help information.
2. Display bind file header.
3. Display SQL statements.
4. Display host variable declarations
5. The name of the bind file.

For more information on db2bfd, refer to the *Command Reference*.

Application, Bind File, and Package Relationships

A package is an object stored in the database that includes information needed to execute specific SQL statements in a single source file. A database application uses one package for every precompiled source file used to build the application. Each package is a separate entity, and has no relationship to any other packages used by the same or other applications. Packages are created by running the precompiler against a source file with binding enabled, or by running the binder at a later time with one or more bind files.

Database applications use packages for some of the same reasons that applications are compiled: improved performance and compactness. By precompiling an SQL statement, the statement is compiled into the package when the application is built, instead of at run time. Each statement is parsed, and a more efficiently interpreted operand string is stored in the package. At run time, the code generated by the precompiler calls run-time services database manager APIs with any variable information required for input or output data, and the information stored in the package is executed.

The advantages of precompilation apply only to static SQL statements. SQL statements that are executed dynamically (using PREPARE and EXECUTE or

EXECUTE IMMEDIATE) are not precompiled; therefore, they must go through the entire set of processing steps at run time.

Note: Do not assume that a static version of an SQL statement automatically executes faster than the same statement processed dynamically. In some cases, static SQL is faster because of the overhead required to prepare the dynamic statement. In other cases, the same statement prepared dynamically executes faster, because the optimizer can make use of current database statistics, rather than the database statistics available at an earlier bind time. Note that if your transaction takes less than a couple of seconds to complete, static SQL will generally be faster. To choose which method to use, you should prototype both forms of binding. For a detailed comparison of static and dynamic SQL, see “Comparing Dynamic SQL with Static SQL” on page 130.

Timestamps

When generating a package or a bind file, the precompiler generates a timestamp. The timestamp is stored in the bind file or package and in the modified source file.

When an application is precompiled with binding enabled, the package and modified source file are generated with timestamps that match. When the application is run, the timestamps are checked for equality. An application and an associated package must have matching timestamps for the application to run, or an SQL0818N error is returned to the application.

Remember that when you bind an application to a database, the first eight characters of the application name are used as the package name *unless you override the default by using the PACKAGE USING option on the PREP command*. This means that if you precompile and bind two programs using the same name, the second will override the package of the first. When you run the first program, you will get a timestamp error because the timestamp for the modified source file no longer matches that of the package in the database.

When an application is precompiled with binding deferred, one or more bind files and modified source files are generated with matching timestamps. To run the application, the bind files produced by the application modules can execute. The binding process must be done for each bind file as discussed in “Binding” on page 51.

The application and package timestamps match because the bind file contains the same timestamp as the one that was stored in the modified source file during precompilation.

Rebinding

Rebinding is the process of recreating a package for an application program that was previously bound. You must rebind packages if they have been marked invalid or inoperative. In some situations, however, you may want to rebind packages that are valid. For example, you may want to take advantage of a newly created index, or make use of updated statistics after executing the RUNSTATS command.

Packages can be dependent on certain types of database objects such as tables, views, aliases, indexes, triggers, referential constraints and table check constraints. If a package is dependent on a database object (such as a table, view, trigger, and so on), and that object is dropped, the package is placed into an *invalid* state. If the object that is dropped is a UDF, the package is placed into an *inoperative* state. For more information, refer to the *Administration Guide, Design and Implementation*.

Invalid packages are implicitly (or automatically) rebound by the database manager when they are executed. Inoperative packages must be explicitly rebound by executing either the BIND command or the REBIND command. Note that implicit rebinding can cause unexpected errors if the implicit rebind fails. That is, the implicit rebind error is returned on the statement being executed which may not be the statement that is actually in error. If an attempt is made to execute an inoperative package, an error occurs. You may decide to explicitly rebind invalid packages rather than have the system automatically rebind them. This enables you to control when the rebinding occurs.

The choice of which command to use to explicitly rebind a package depends on the circumstances. You must use the BIND command to rebind a package for a program which has been modified to include more, fewer, or changed SQL statements. You must also use the BIND command if you need to change any bind options from the values with which the package was originally bound. In all other cases, use either the BIND or REBIND command. You should use REBIND whenever your situation does not specifically require the use of BIND, as the performance of REBIND is significantly better than that of BIND.

For details on the REBIND command, refer to the *Command Reference*.

Chapter 4. Writing Static SQL Programs

Characteristics and Reasons for Using Static SQL	60	FORTTRAN Example: OPENFTCH.SQF	99
Advantages of Static SQL	60	Advanced Scrolling Techniques	101
Example: Static SQL Program	61	Scrolling Through Data that has Already Been Retrieved	101
How the Static Program Works	62	Keeping a Copy of the Data	101
C Example: STATIC.SQC	64	Retrieving the Data a Second Time	102
Java Example: Static.sqlj	65	Retrieving from the Beginning	102
COBOL Example: STATIC.SQB	66	Retrieving from the Middle	102
FORTTRAN Example: STATIC.SQF	67	Order of Rows in the Second Result Table	102
Coding SQL Statements to Retrieve and Manipulate Data	68	Retrieving in Reverse Order	103
Retrieving Data	68	Establishing a Position at the End of a Table	104
Using Host Variables	69	Updating Previously Retrieved Data	104
Declaration Generator - db2dclgn	71	Example: UPDAT Program	104
Using Indicator Variables	73	How the UPDAT Program Works	105
Data Types	75	C Example: UPDAT.SQC	107
Using an Indicator Variable in the STATIC program	78	Java Example: Updat.sqlj	109
Selecting Multiple Rows Using a Cursor	78	COBOL Example: UPDAT.SQB	111
Declaring and Using the Cursor	79	FORTTRAN Example: UPDAT.SQF	113
Cursors and Unit of Work	80	REXX Example: UPDAT.CMD	115
Considerations	80	Diagnostic Handling and the SQLCA Structure	117
Read Only Cursors	80	Return Codes	117
WITH HOLD Option	80	SQLCODE and SQLSTATE	117
Example: Cursor Program	82	Token Truncation in SQLCA Structure	118
How the Cursor Program Works	82	Handling Errors using the WHENEVER Statement	118
C Example: CURSOR.SQC	83	Exception, Signal, Interrupt Handler	119
Java Example: Cursor.sqlj	85	Considerations	119
COBOL Example: CURSOR.SQB	87	Exit List Routine Considerations	120
FORTTRAN Example: CURSOR.SQF	89	Using GET ERROR MESSAGE in Example Programs	120
Updating and Deleting Retrieved Data	90	C Example: UTIL.C	120
Updating Retrieved Data	90	Java Example: Catching SQLException	123
Deleting Retrieved Data	90	COBOL Example: CHECKERR.CBL	124
Types of Cursors	90	FORTTRAN Example: UTIL.F	126
Example: OPENFTCH Program	91	REXX Example: CHECKERR Procedure	128
How the OPENFTCH Program Works	91		
C Example: OPENFTCH.SQC	93		
Java Example: Openftch.sqlj	95		
COBOL Example: OPENFTCH.SQB	97		

Characteristics and Reasons for Using Static SQL

When the syntax of embedded SQL statements is fully known at precompile time, the statements are referred to as *static SQL*. This is in contrast to *dynamic SQL* statements whose syntax is not known until run time.

Note: Static SQL is not supported in interpreted languages, such as REXX.

The structure of an SQL statement must be completely specified in order for a statement to be considered static. For example, the names for the columns and tables referenced in a statement must be fully known at precompile time. The only information that can be specified at run time are values for any host variables referenced by the statement. However, host variable information, such as data types, must still be precompiled.

When a static SQL statement is prepared, an executable form of the statement is created and stored in the package in the database. The executable form can be constructed either at precompile time, or at a later bind time. In either case, preparation occurs *before* run time. The authorization of the person binding the application is used, and optimization is based upon database statistics and configuration parameters that may not be current when the application runs.

Advantages of Static SQL

Programming using static SQL requires less effort than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager run-time services API calls that the host language compiler can process.

Because the authorization of the person binding the application is used, the end user does not require direct privileges to execute the statements in the package. For example, an application could allow a user to update parts of a table without granting an update privilege on the entire table. This can be achieved by restricting the static SQL statements to allow updates only to certain columns or a range of values.

Static SQL statements are *persistent*, meaning that the statements last for as long as the package exists. Dynamic SQL statements are cached until they are either invalidated, freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the DB2 SQL compiler whenever a cached statement becomes invalid. For information on caching and the reasons for invalidation of a cached statement, refer to the *SQL Reference*.

The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements cease to exist when this occurs. In addition, static SQL does not have to be compiled by the DB2 SQL compiler at run time, while dynamic SQL must be explicitly compiled at run time (for example, by using the PREPARE statement). Because DB2 caches dynamic SQL statements, the statements do not need to be compiled often by DB2, but they must be compiled at least once when you execute the application.

There can be performance advantages to static SQL. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically since the overhead of preparing an executable form of the statement is done at precompile time instead of at run time.

Note: The performance of static SQL depends on the statistics of the database the last time the application was bound. However, if these statistics change, the performance of equivalent dynamic SQL can be very different. If, for example, an index is added to a database at a later time, an application using static SQL cannot take advantage of the index unless it is re-bound to the database. In addition, if you are using host variables in a static SQL statement, the optimizer will not be able to take advantage of any distribution statistics for the table.

Example: Static SQL Program

This sample program shows examples of static SQL statements and database manager API calls in the following supported languages:

C	static.sqc
Java	Static.sqlj
COBOL	static.sqb
FORTRAN	static.sqf

The REXX language does not support static SQL, so a sample is not provided.

This sample program contains a query that selects a single row. Such a query can be performed using the SELECT INTO statement.

The SELECT INTO statement selects one row of data from tables in a database, and the values in this row are assigned to host variables specified in the statement. Host variables are discussed in detail in “Using Host Variables” on page 69. For example, the following statement will deliver the salary of the employee with the last name of 'HAAS' into the host variable empsal:

```

SELECT SALARY
  INTO :empsal
  FROM EMPLOYEE
 WHERE LASTNAME='HAAS'

```

A SELECT INTO statement must be specified to return only one or zero rows. Finding more than one row results in an error, SQLCODE -811 (SQLSTATE 21000). If several rows can be the result of a query, a cursor must be used to process the rows. See “Selecting Multiple Rows Using a Cursor” on page 78 for more information.

For more details on the SELECT INTO statement, refer to the *SQL Reference*.

For an introductory discussion on how to write SELECT statements, see “Coding SQL Statements to Retrieve and Manipulate Data” on page 68.

How the Static Program Works

1. **Include the SQLCA.** The INCLUDE SQLCA statement defines and declares the SQLCA structure, and defines SQLCODE and SQLSTATE as elements within the structure. The SQLCODE field of the SQLCA structure is updated with diagnostic information by the database manager after every execution of SQL statements or database manager API calls.
2. **Declare host variables.** The SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. These are variables that can be referenced in SQL statements. Host variables are used to pass data to the database manager or to hold data returned by it. They are prefixed with a colon (:) when referenced in an SQL statement. For more information, see “Using Host Variables” on page 69.
3. **Connect to database.** The program connects to the sample database, and requests shared access to it. (It is assumed that a START DATABASE MANAGER API call or db2start command has been issued.) Other programs that connect to the same database using shared access are also granted access.
4. **Retrieve data.** The SELECT INTO statement retrieves a single value based upon a query. This example retrieves the FIRSTNME column from the EMPLOYEE table where the value of the LASTNAME column is JOHNSON. The value SYBIL is returned and placed in the host variable firstname. The sample tables supplied with DB2 are listed in the appendix of the *SQL Reference*.
5. **Process errors.** The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C	check_error is redefined as CHECKERR and is located in the util.c file.
Java	Any SQL error is thrown as an SQLException and handled in the catch block of the application.
COBOL	CHECKERR is an external program named checkerr.cbl
FORTTRAN	CHECKERR is a subroutine located in the util.f file.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

6. **Disconnect from database.** The program disconnects from the database by executing the CONNECT RESET statement. Note that SQLJ programs automatically close the database connection when the program returns.

C Example: STATIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA; 1
#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {
    EXEC SQL BEGIN DECLARE SECTION; 2
        char firstname[13];
        char userid[9];
        char passwd[19];
    EXEC SQL END DECLARE SECTION;

#ifdef DB268K
    /* Before making any API calls for 68K environment,
       need to initial the Library Manager */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
#endif

    printf( "Sample C program: STATIC\n" );
    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd; 3
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: static [userid passwd]\n\n");
        return 1;
    } /* endif */

    EXEC SQL SELECT FIRSTNME INTO :firstname 4
        FROM employee
        WHERE LASTNAME = 'JOHNSON';
    CHECKERR ("SELECT statement"); 5
    printf( "First name = %s\n", firstname );
    EXEC SQL CONNECT RESET; 6
    CHECKERR ("CONNECT RESET");
    return 0;
}
/* end of program : STATIC.SQC */
```

Java Example: Static.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Static
{
    static
    {
        try
        {
            Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {
            System.out.println ("\n Error loading DB2 Driver...\n");
            System.out.println (e);
            System.exit(1);
        }
    }

    public static void main(String argv[])
    {
        try
        {
            System.out.println (" Java Static Sample");

            String url = "jdbc:db2:sample";          // URL is jdbc:db2:dbname
            Connection con = null;

            // Set the connection 3
            if (argv.length == 0)
            {
                // connect with default id/password
                con = DriverManager.getConnection(url);
            }
            else if (argv.length == 2)
            {
                String userid = argv[0];
                String passwd = argv[1];
                // connect with user-provided username and password
                con = DriverManager.getConnection(url, userid, passwd);
            }
            else
            {
                throw new Exception("\nUsage: java Static [username password]\n");
            }

            // Set the default context
            DefaultContext ctx = new DefaultContext(con);
            DefaultContext.setDefaultContext(ctx);

            String firstname = null; 2
            #sql { SELECT FIRSTNME INTO :firstname
                  FROM employee
                  WHERE LASTNAME = 'JOHNSON' } ; 4

            System.out.println ("First name = " + firstname);
        }
        catch( Exception e ) 5
        {
            System.out.println (e);
        }
    }
}
```

COBOL Example: STATIC.SQB

Identification Division.
Program-ID. "static".

Data Division.

Working-Storage Section.

```
    copy "sql.cbl".  
    copy "sqlca.cbl".  
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
01 firstname      pic x(12).  
01 userid         pic x(8).  
01 passwd.  
    49 passwd-length pic s9(4) comp-5 value 0.  
    49 passwd-name  pic x(18).  
    EXEC SQL END DECLARE SECTION END-EXEC.  
  
77 errloc         pic x(80).
```

1
2

Procedure Division.

Main Section.

```
    display "Sample COBOL program: STATIC".  
    display "Enter your user id (default none): "  
        with no advancing.  
    accept userid.  
  
    if userid = spaces  
        EXEC SQL CONNECT TO sample END-EXEC  
    else  
        display "Enter your password : " with no advancing  
        accept passwd-name.
```

- * Passwords in a CONNECT statement must be entered in a VARCHAR format
- * with the length of the input string.
 inspect passwd-name tallying passwd-length for characters
 before initial " ".

```
    EXEC SQL CONNECT TO sample USER :userid USING :passwd  
        END-EXEC.  
    move "CONNECT TO" to errloc.  
    call "checkerr" using SQLCA errloc.
```

3

```
    EXEC SQL SELECT FIRSTNME INTO :firstname  
        FROM EMPLOYEE  
        WHERE LASTNAME = 'JOHNSON' END-EXEC.  
    move "SELECT" to errloc.  
    call "checkerr" using SQLCA errloc.  
    display "First name = ", firstname.
```

4

5

```
    EXEC SQL CONNECT RESET END-EXEC.  
    move "CONNECT RESET" to errloc.  
    call "checkerr" using SQLCA errloc.
```

6

End-Prog.
stop run.

FORTRAN Example: STATIC.SQF

```
program static
implicit none

include 'sqlenv.f'
EXEC SQL INCLUDE SQLCA 1

EXEC SQL BEGIN DECLARE SECTION 2
  character*12  firstname
  character*8   userid
  character*18  passwd
EXEC SQL END DECLARE SECTION

character*80    errloc

print *, 'Sample Fortran Program: STATIC'

print *, 'Enter your user id (default none):'
read 100, userid
100 format (a8)

if( userid(1:1) .eq. ' ' ) then
  EXEC SQL CONNECT TO sample
else
print *, 'Enter your password :'
read 100, passwd
EXEC SQL CONNECT TO sample USER :userid USING :passwd
end if 3
errloc = 'CONNECT'
call checkerr (sqlca, errloc, *999)

EXEC SQL SELECT firstnme INTO :firstname
c FROM employee WHERE lastname = 'JOHNSON' 4
errloc = 'SELECT'
call checkerr (sqlca, errloc, *999) 5

print *, 'First name = ', firstname

EXEC SQL CONNECT RESET 6
errloc = 'CONNECT RESET'
call checkerr (sqlca, errloc, *999)

999 stop
end
```

Coding SQL Statements to Retrieve and Manipulate Data

The database manager provides application programmers with statements for retrieving and manipulating data; the coding task consists of embedding these statements into the host language code. This section shows how to code statements that will retrieve and manipulate data for one or more rows of data in DB2 tables. (It does not go into the details of the different host languages.) For the exact rules of placement, continuation, and delimiting SQL statements, see:

- “Chapter 16. Programming in C and C++” on page 491
- “Chapter 17. Programming in Java” on page 533
- “Chapter 18. Programming in COBOL” on page 569
- “Chapter 19. Programming in FORTRAN” on page 591
- “Chapter 20. Programming in REXX” on page 607.

Retrieving Data

One of the most common tasks of an SQL application program is to retrieve data. This is done using the *select-statement*, which is a form of query that searches for rows of tables in the database that meet specified search conditions. If such rows exist, the data is retrieved and put into specified variables in the host program, where it can be used for whatever it was designed to do.

After you have written a *select-statement*, you code the SQL statements that define how information will be passed to your application.

You can think of the result of a select-statement as being a table having rows and columns, much like a table in the database. If only one row is returned, you can deliver the results directly into host variables specified by the *SELECT INTO* statement.

If more than one row is returned, you must use a *cursor* to fetch them one at a time. A cursor is a named control structure used by an application program to point to a specific row within an ordered set of rows. For information about how to code and use cursors, see the following sections:

- “Declaring and Using the Cursor” on page 79,
- “Selecting Multiple Rows Using a Cursor” on page 78,
- “Example: Cursor Program” on page 82.

Using Host Variables

Host variables are variables referenced by embedded SQL statements. They transmit data between the database manager and an application program. When you use a host variable in an *SQL statement*, you must prefix its name with a colon, (:). When you use a host variable in a *host language statement*, omit the colon.

Host variables are declared in compiled host languages, and are delimited by BEGIN DECLARE SECTION and END DECLARE SECTION statements. These statements enable the precompiler to find the declarations.

Note: Java JDBC and SQLJ programs do not use declare sections. Host variables in Java follow the normal Java variable declaration syntax.

Host variables are declared using a subset of the host language. For a description of the supported syntax for your host language, see:

- “Chapter 16. Programming in C and C++” on page 491
- “Chapter 17. Programming in Java” on page 533
- “Chapter 18. Programming in COBOL” on page 569
- “Chapter 19. Programming in FORTRAN” on page 591
- “Chapter 20. Programming in REXX” on page 607.

The following rules apply to host variable declaration sections:

- All host variables must be declared in the source file before they are referenced, except for host variables referring to SQLDA structures.
- Multiple declare sections may be used in one source file.
- The precompiler is unaware of host language variable scoping rules.

With respect to SQL statements, all host variables have a global scope regardless of where they are actually declared in a single source file. Therefore, host variable names must be unique within a source file.

This does not mean that the DB2 precompiler changes the scope of host variables to global so that they can be accessed outside the scope in which they are defined. Consider the following example:

```
foo1(){
    .
    .
    .
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
    x=10;
    .
    .
}
```

```

}
.
foo2(){
.
.
.
y=x;
.
.
.
}

```

Depending on the language, the above example will either fail to compile because variable `x` is not declared in function `foo2()` or the value of `x` would not be set to 10 in `foo2()`. To avoid this problem, you must either declare `x` as a global variable, or pass `x` as a parameter to function `foo2()` as follows:

```

foo1(){
.
.
.
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
    x=10;
    foo2(x);
.
.
.
}

foo2(int x){
.
.
.
    y=x;
.
.
.
}

```

For further information on declaring host variables, see:

- “Declaration Generator - db2dclgn” on page 71 to generate host variable declaration source code automatically with the `db2dclgn` tool
- Table 4 on page 72 for examples of how host variables appear in source code

- Table 5 on page 72 for examples of how to reference host variables in the supported host languages
- “Naming Host Variables” on page 611 and “Referencing Host Variables” on page 611 for information on naming and referencing host variables in REXX

Declaration Generator - db2dclgn

The Declaration Generator speeds application development by generating declarations for a given table in a database. It creates embedded SQL declaration source files which you can easily insert into your applications. db2dclgn supports the C/C++, Java, COBOL, and FORTRAN languages.

To generate declaration files, enter the db2dclgn command in the following format:

```
db2dclgn -d database-name -t table-name [options]
```

For example, to generate the declarations for the STAFF table in the SAMPLE database in C in the output file staff.h, issue the following command:

```
db2dclgn -d sample -t staff -l C
```

The resulting staff.h file contains:

```
struct
{
    short id;
    struct
    {
        short length;
        char data[9];
    } name;
    short dept;
    char job[5];
    short years;
    double salary;
    double comm;
} staff;
```

For detailed information on db2dc1gn, refer to the *Command Reference*.

Table 4. Declaring Host Variables

Language	Example Source Code
C/C++	<pre>EXEC SQL BEGIN DECLARE SECTION; short dept=38, age=26; double salary; char CH; char name1[9], NAME2[9]; /* C comment */ short nul_ind; EXEC SQL END DECLARE SECTION;</pre>
Java	<pre>// Note that Java host variable declarations follow // normal Java variable declaration rules, and have // no equivalent of a DECLARE SECTION short dept=38, age=26; double salary; char CH; String name1[9], NAME2[9]; /* Java comment */ short nul_ind;</pre>
COBOL	<pre>EXEC SQL BEGIN DECLARE SECTION END-EXEC. 01 age PIC S9(4) COMP-5 VALUE 26. 01 DEPT PIC S9(9) COMP-5 VALUE 38. 01 salary PIC S9(6)V9(3) COMP-3. 01 CH PIC X(1). 01 name1 PIC X(8). 01 NAME2 PIC X(8). * COBOL comment 01 nul-ind PIC S9(4) COMP-5. EXEC SQL END DECLARE SECTION END-EXEC.</pre>
FORTRAN	<pre>EXEC SQL BEGIN DECLARE SECTION integer*2 age /26/ integer*4 dept /38/ real*8 salary character ch character*8 name1,NAME2 C FORTRAN comment integer*2 nul_ind EXEC SQL END DECLARE SECTION</pre>

Table 5. Referencing Host Variables

Language	Example Source Code
C/C++	<pre>EXEC SQL FETCH C1 INTO :cm; printf("Commission = %f\n", cm);</pre>
JAVA (SQLJ)	<pre>#SQL { FETCH :c1 INTO :cm }; System.out.println("Commission = " + cm);</pre>

Table 5. Referencing Host Variables (continued)

Language	Example Source Code
COBOL	EXEC SQL FETCH C1 INTO :cm END-EXEC DISPLAY 'Commission = ' cm
FORTRAN	EXEC SQL FETCH C1 INTO :cm WRITE(*,*) 'Commission = ', cm

Using Indicator Variables

Applications written in languages other than Java must prepare for receiving null values by associating an *indicator variable* with any host variable that can receive a null. Java applications compare the value of the host variable with Java null to determine whether the received value is null. An indicator variable is shared by both the database manager and the host application; therefore, the indicator variable must be declared in the application as a host variable. This host variable corresponds to the SQL data type SMALLINT.

An indicator variable is placed in an SQL statement immediately after the host variable, and is prefixed with a colon. A space can separate the indicator variable from the host variable, but is not required. However, do not put a comma between the host variable and the indicator variable. You can also specify an indicator variable by using the optional INDICATOR keyword, which you place between the host variable and its indicator.

Indicator Variables shows indicator variable usage in the supported host languages using the INDICATOR keyword.

Language

Example Source Code

C/C++

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;
if ( cmind < 0 )
    printf( "Commission is NULL\n" );
```

Java (SQLJ)

```
#SQL { FETCH :c1 INTO :cm };
if ( cm == null )
    System.out.println( "Commission is NULL\n" );
```

COBOL

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC
IF cmind LESS THAN 0
    DISPLAY 'Commission is NULL'
```

FORTRAN

```
EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind
IF ( cmind .LT. 0 ) THEN
    WRITE(*,*) 'Commission is NULL'
ENDIF
```

In the figure, *cmind* is examined for a negative value. If it is not negative, the application can use the returned value of *cm*. If it is negative, the fetched value is NULL and *cm* should not be used. The database manager does not change the value of the host variable in this case.

Note: If the database configuration parameter DFT_SQLMATHWARN is set to 'YES', the value of *cmind* may be -2. This indicates a NULL that was caused by evaluating an expression with an arithmetic error or by an overflow while attempting to convert the numeric result value to the host variable.

If the data type can handle NULLs, the application must provide a NULL indicator. Otherwise, an error may occur. If a NULL indicator is not used, an SQLCODE -305 (SQLSTATE 22002) is returned.

If the SQLCA structure indicates a truncation warning, the indicator variables can be examined for truncation. If an indicator variable has a positive value, a truncation occurred.

- If the seconds portion of a TIME data type is truncated, the indicator value contains the seconds portion of the truncated data.
- For all other string data types, except large objects (LOB), the indicator value represents the actual length of the data returned. User-defined distinct types (UDT) are handled in the same way as their base type.

When processing INSERT or UPDATE statements, the database manager checks the indicator variable if one exists. If the indicator variable is negative, the database manager sets the target column value to NULL if NULLs are allowed. If the indicator variable is zero or positive, the database manager uses the value of the associated host variable.

The SQLWARN1 field in the SQLCA structure may contain an 'X' or 'W' if the value of a string column is truncated when it is assigned to a host variable. It contains an 'N' if a null terminator is truncated.

A value of 'X' is returned by the database manager only if all of the following conditions are met:

- A mixed code page connection exists where conversion of character string data from the database code page to the application code page involves a change in the length of the data.
- A cursor is blocked.

- An indicator variable is provided by your application.

The value returned in the indicator variable will be the length of the resultant character string in the application's code page.

In all other cases involving data truncation, (as opposed to NULL terminator truncation), the database manager returns a 'W'. In this case, the database manager returns a value in the indicator variable to the application that is the length of the resultant character string in the code page of the select list item (either the application code page, the data base code page, or nothing). For related information, refer to the *SQL Reference*.

Data Types

Each column of every DB2 table is given an *SQL data type* when the column is created. For information about how these types are assigned to columns, refer to the CREATE TABLE statement in the *SQL Reference*. The database manager supports the following column data types:

SMALLINT

16-bit signed integer.

INTEGER

32-bit signed integer. **INT** can be used as a synonym for this type.

BIGINT

64-bit signed integer.

DOUBLE

Double-precision floating point. **DOUBLE PRECISION** and **FLOAT(n)** (where n is greater than 24) are synonyms for this type.

REAL Single-precision floating point. **FLOAT(n)** (where n is less than 24) is a synonym for this type.

DECIMAL

Packed decimal. **DEC**, **NUMERIC**, and **NUM** are synonyms for this type.

CHAR

Fixed-length character string of length 1 byte to 254 bytes.
CHARACTER can be used as a synonym for this type.

VARCHAR

Variable-length character string of length 1 byte to 32672 bytes.
CHARACTER VARYING and **CHAR VARYING** are synonyms for this type.

LONG VARCHAR

Long variable-length character string of length 1 byte to 32 700 bytes.

CLOB Large object variable-length character string of length 1 byte to 2 gigabytes.

BLOB Large object variable-length binary string of length 1 byte to 2 gigabytes.

DATE Character string of length 10 representing a date.

TIME Character string of length 8 representing a time.

TIMESTAMP

Character string of length 26 representing a timestamp.

The following data types are supported only in double-byte character set (DBCS) and Extended UNIX Code (EUC) character set environments:

GRAPHIC

Fixed-length graphic string of length 1 to 127 double-byte characters.

VARGRAPHIC

Variable-length graphic string of length 1 to 16336 double-byte characters.

LONG VARGRAPHIC

Long variable-length graphic string of length 1 to 16 350 double-byte characters.

DBCLOB

Large object variable-length graphic string of length 1 to 1 073 741 823 double-byte characters.

Notes:

1. Every supported data type can have the NOT NULL attribute. This is treated as another type.
2. The above set of data types can be extended by defining user-defined distinct types (UDT). UDTs are separate data types which use the representation of one of the built-in SQL types.

Supported host languages have data types that correspond to the majority of the database manager data types. Only these host language data types can be used in host variable declarations. When the precompiler finds a host variable declaration, it determines the appropriate SQL data type value. The database manager uses this value to convert the data exchanged between itself and the application.

As the application programmer, it is important for you to understand how the database manager handles comparisons and assignments between different data types. Simply put, data types must be compatible with each other during

assignment and comparison operations, whether the database manager is working with two SQL column data types, two host-language data types, or one of each.

The *general* rule for data type compatibility is that all supported host-language numeric data types are comparable and assignable with all database manager numeric data types, and all host-language character types are compatible with all database manager character types; numeric types are incompatible with character types. However, there are also some exceptions to this general rule depending on host language idiosyncrasies and limitations imposed when working with large objects.

Within SQL statements, DB2 provides conversions between compatible data types. For example, in the following SELECT statement, SALARY and BONUS are DECIMAL columns; however, each employee's total compensation is returned as DOUBLE data:

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

Note that the execution of the above statement includes conversion between DECIMAL and DOUBLE data types. To make the query results more readable on your screen, you could use the following SELECT statement:

```
SELECT EMPNO, DIGIT(SALARY+BONUS) FROM EMPLOYEE
```

To convert data within your application, contact your compiler vendor for additional routines, classes, built-in types, or APIs that supports this conversion.

Character data types may also be subject to character conversion. If your application code page is not the same as your database code page, see "Conversion Between Different Code Pages" on page 407.

For the list of supported SQL data types and the corresponding host language data types, see the following:

- for C/C++, "Supported SQL Data Types" on page 525
- for Java, "Supported SQL Data Types" on page 535
- for COBOL, "Supported SQL Data Types" on page 586
- for FORTRAN, "Supported SQL Data Types" on page 602
- for REXX, "Supported SQL Data Types" on page 617.

For more information about SQL data types, the rules of assignments and comparisons, and data conversion and conversion errors, refer to the *SQL Reference*.

Using an Indicator Variable in the STATIC program

The following code segments show the modification to the corresponding segments in the C version of the sample STATIC program, listed in “C Example: STATIC.SQC” on page 64. They show the implementation of indicator variables on data columns that are nullable. In this example, the STATIC program is extended to select another column, WORKDEPT. This column can have a null value. An indicator variable needs to be declared as a host variable before being used.

```
.
.
.
EXEC SQL BEGIN DECLARE SECTION;
char wd[3];
short wd_ind;
char firstname[13];
.
.
.

EXEC SQL END DECLARE SECTION;
.
.
.
/* CONNECT TO SAMPLE DATABASE */
.
.
.
EXEC SQL SELECT FIRSTNME, WORKDEPT INTO :firstname, :wd:wdind
FROM EMPLOYEE
WHERE LASTNAME = 'JOHNSON';
.
.
.
```

Selecting Multiple Rows Using a Cursor

To allow an application to retrieve a set of rows, SQL uses a mechanism called a *cursor*.

To help understand the concept of a cursor, assume that the database manager builds a *result table* to hold all the rows retrieved by executing a SELECT statement. A cursor makes rows from the result table available to an application, by identifying or pointing to a *current row* of this table. When a cursor is used, an application can retrieve each row sequentially from the result table until an end of data condition, that is, the NOT FOUND condition, SQLCODE +100 (SQLSTATE 02000) is reached. The set of rows

obtained as a result of executing the SELECT statement can consist of zero, one, or more rows, depending on the number of rows that satisfy the search condition.

The steps involved in processing a cursor are as follows:

1. Specify the cursor using a DECLARE CURSOR statement.
2. Perform the query and build the result table using the OPEN statement.
3. Retrieve rows one at a time using the FETCH statement.
4. Process rows with the DELETE or UPDATE statements (if required).
5. Terminate the cursor using the CLOSE statement.

An application can use several cursors concurrently. Each cursor requires its own set of DECLARE CURSOR, OPEN, CLOSE, and FETCH statements.

See “Example: Cursor Program” on page 82 for an example of how an application can select a set of rows and, using a cursor, process the set one row at a time.

Declaring and Using the Cursor

The DECLARE CURSOR statement defines and names the cursor, identifying the set of rows to be retrieved using a SELECT statement.

The application assigns a name for the cursor. This name is referred to in subsequent OPEN, FETCH, and CLOSE statements. The query is any valid select statement.

Declare Cursor Statement shows a DECLARE statement associated with a static SELECT statement.

Language

Example Source Code

C/C++

```
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT PNAME, DEPT FROM STAFF
    WHERE JOB=:host_var;
```

Java (SQLJ)

```
#sql iterator cursor1(host_var data type);
#sql cursor1 = { SELECT PNAME, DEPT FROM STAFF
    WHERE JOB=:host_var };
```

COBOL

```
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT NAME, DEPT FROM STAFF
    WHERE JOB=:host-var END-EXEC.
```

FORTRAN

```
EXEC SQL DECLARE C1 CURSOR FOR
+ SELECT NAME, DEPT FROM STAFF
+ WHERE JOB=:host_var
```

Note: The placement of the DECLARE statement is arbitrary, but it must be placed above the first use of the cursor.

Cursors and Unit of Work Considerations

The actions of a COMMIT or ROLLBACK operation vary for cursors, depending on how the cursors are declared.

Read Only Cursors

If a cursor is determined to be read only and uses a repeatable read isolation level, repeatable read locks are still gathered and maintained on system tables needed by the unit of work. Therefore, it is important for applications to periodically issue COMMIT statements, even for read only cursors.

WITH HOLD Option

If an application completes a unit of work by issuing a COMMIT statement, *all open cursors*, except those declared using the WITH HOLD option, are automatically closed by the database manager.

A cursor that is declared WITH HOLD maintains the resources it accesses across multiple units of work. The exact effect of declaring a cursor WITH HOLD depends on how the unit of work ends.

If the unit of work ends with a COMMIT statement, open cursors defined WITH HOLD remain OPEN. The cursor is positioned before the next logical row of the result table. In addition, prepared statements referencing OPEN cursors defined WITH HOLD are retained. Only FETCH and CLOSE requests associated with a particular cursor are valid immediately following the COMMIT. UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF statements are valid only for rows fetched within the same unit of work. If a package is rebound during a unit of work, all held cursors are closed.

If the unit of work ends with a ROLLBACK statement, all open cursors are closed, all locks acquired during the unit of work are released, and all prepared statements that are dependent on work done in that unit are dropped.

For example, suppose that the TEMPL table contains 1000 entries. You want to update the salary column for all employees, and you expect to issue a COMMIT statement every time you update 100 rows.

1. Declare the cursor using the WITH HOLD option:

```
EXEC SQL DECLARE EMPLUPDT CURSOR WITH HOLD FOR
  SELECT EMPNO, LASTNAME, PHONENO, JOBCODE, SALARY
  FROM TEMPL FOR UPDATE OF SALARY
```

2. Open the cursor and fetch data from the result table one row at a time:

```
EXEC SQL OPEN EMPLUPDT
```

```
.
.
.
```

```
EXEC SQL FETCH EMPLUPDT
  INTO :upd_emp, :upd_lname, :upd_tele, :upd_jobcd, :upd_wage,
```

3. When you want to update or delete a row, use an UPDATE or DELETE statement using the WHERE CURRENT OF option. For example, to update the current row, your program can issue:

```
EXEC SQL UPDATE TEMPL SET SALARY = :newsalary
  WHERE CURRENT OF EMPLUPDT
```

4. After a COMMIT is issued, you must issue a FETCH before you can update another row.

You should include code in your application to detect and handle an SQLCODE -501 (SQLSTATE 24501), which can be returned on a FETCH or CLOSE statement if your application either:

- Uses cursors declared WITH HOLD
- Executes more than one unit of work and leaves a WITH HOLD cursor open across the unit of work boundary (COMMIT WORK).

If an application invalidates its package by dropping a table on which it is dependent, the package gets rebound dynamically. If this is the case, an SQLCODE -501 (SQLSTATE 24501) is returned for a FETCH or CLOSE statement because the database manager closes the cursor. The way to handle an SQLCODE -501 (SQLSTATE 24501) in this situation depends on whether you want to fetch rows from the cursor.

- If you want to fetch rows from the cursor, open the cursor, then run the FETCH statement. Note, however, that the OPEN statement repositions the cursor to the start. The previous position held at the COMMIT WORK statement is lost.
- If you do not want to fetch rows from the cursor, do not issue any more SQL requests against the cursor.

WITH RELEASE Option: When an application closes a cursor using the WITH RELEASE option, DB2 attempts to release all READ locks that the cursor still holds. The cursor will only continue to hold WRITE locks. If the application closes the cursor without using the RELEASE option, the READ and WRITE locks will be released when the unit of work completes.

Example: Cursor Program

This sample program shows the SQL statements that define and use a cursor. The cursor is processed using static SQL. The sample is available in the following programming languages:

C	cursor.sqc
Java	Cursor.sqlj
COBOL	cursor.sqb
FORTRAN	cursor.sqf

Since REXX does not support static SQL, a sample is not provided. See “Example: Dynamic SQL Program” on page 136 for a REXX example that processes a cursor dynamically.

How the Cursor Program Works

1. **Declare the cursor.** The DECLARE CURSOR statement associates the cursor c1 to a query. The query identifies the rows that the application retrieves using the FETCH statement. The job field of staff is defined to be updatable, even though it is not specified in the result table.
2. **Open the cursor.** The cursor c1 is opened, causing the database manager to perform the query and build a result table. The cursor is positioned *before* the first row.
3. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the host variables. This row becomes the *current* row.
4. **Close the cursor.** The CLOSE statement is issued, releasing the resources associated with the cursor. The cursor can be opened again, however.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C	check_error is redefined as CHECKERR and is located in the util.c file.
Java	Any SQL error is thrown as an SQLException and handled in the catch block of the application.
COBOL	CHECKERR is an external program named checkerr.cbl
FORTRAN	CHECKERR is a subroutine located in the util.f file.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

C Example: CURSOR.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR)  if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
        char  pname[10];
        short dept;
        char userid[9];
        char passwd[19];
    EXEC SQL END DECLARE SECTION;

#ifdef DB268K
    /* Before making any API calls for 68K environment,
       need to initial the Library Manager */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
#endif

    printf( "Sample C program: CURSOR \n" );

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: cursor [userid passwd]\n\n");
        return 1;
    } /* endif */

    EXEC SQL DECLARE c1 CURSOR FOR 1
        SELECT name, dept FROM staff WHERE job='Mgr'
        FOR UPDATE OF job;

    EXEC SQL OPEN c1; 2
    CHECKERR ("OPEN CURSOR");
```

```

do {
    EXEC SQL FETCH c1 INTO :pname, :dept; 3
    if (SQLCODE != 0) break;

    printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
} while ( 1 );

EXEC SQL CLOSE c1; 4
CHECKERR ("CLOSE CURSOR");

EXEC SQL ROLLBACK;
CHECKERR ("ROLLBACK");
printf( "\n\n second thought -- changes rolled back.\n" );

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : CURSOR.SQC */

```


Java Example: Cursor.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator CursorByName(String name, short dept) ;
#sql iterator CursorByPos(String, short ) ;

class Cursor
{
    static
    {
        try
        {
            Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {
            System.out.println ("\n  Error loading DB2 Driver...\n");
            System.out.println (e);
            System.exit(1);
        }
    }
}

public static void main(String argv[])
{
    try
    {
        System.out.println ("  Java Cursor Sample");

        String url = "jdbc:db2:sample";
        // URL is jdbc:db2:dbname
        Connection con = null;

        // Set the connection
        if (argv.length == 0)
        {
            // connect with default id/password
            con = DriverManager.getConnection(url);
        }
        else if (argv.length == 2)
        {
            String userid = argv[0];
            String passwd = argv[1];

            // connect with user-provided username and password
            con = DriverManager.getConnection(url, userid, passwd);
        }
        else
        {
            throw new Exception("Usage: java Cursor [username password]");
        }

        // Set the default context
        DefaultContext ctx = new DefaultContext(con);
        DefaultContext.setDefaultContext(ctx);

        // Enable transactions
        con.setAutoCommit(false);

        // Using cursors
        try
        {
            CursorByName cursorByName;
            CursorByPos cursorByPos;
        }
    }
}
```

```

String name = null;
short dept=0;

// Using the JDBC ResultSet cursor method
System.out.println("\nUsing the JDBC ResultSet cursor method");
System.out.println(" with a 'bind by name' cursor ...\n");

#sql cursorByName = {
    SELECT name, dept FROM staff
    WHERE job='Mgr' FOR UPDATE OF job }; 1
while (cursorByName.next()) 2
{   name = cursorByName.name(); 3
    dept = cursorByName.dept();

    System.out.print (" name= " + name);
    System.out.print (" dept= " + dept);
    System.out.print ("\n");
}
cursorByName.close(); 4

// Using the SQLJ iterator cursor method
System.out.println("\nUsing the SQLJ iterator cursor method");
System.out.println(" with a 'bind by position' cursor ...\n");

#sql cursorByPos = {
    SELECT name, dept FROM staff
    WHERE job='Mgr' FOR UPDATE OF job }; 1 2
while (true)
{   #sql { FETCH :cursorByPos INTO :name, :dept }; 3
    if (cursorByPos.endFetch()) break;

    System.out.print (" name= " + name);
    System.out.print (" dept= " + dept);
    System.out.print ("\n");
}
cursorByPos.close(); 4
}
catch( Exception e )
{   throw e;
}
finally
{   // Rollback the transaction
    System.out.println("\nRollback the transaction...");
    #sql { ROLLBACK };
    System.out.println("Rollback done.");
}
}
catch( Exception e )
{   System.out.println (e);
}
}
}

```

COBOL Example: CURSOR.SQB

Identification Division.
Program-ID. "cursor".

Data Division.
Working-Storage Section.

copy "sqlenv.cbl".
copy "sql.cbl".
copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 pname pic x(10).
77 dept pic s9(4) comp-5.
01 userid pic x(8).
01 passwd.
49 passwd-length pic s9(4) comp-5 value 0.
49 passwd-name pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc pic x(80).

Procedure Division.
Main Section.

display "Sample COBOL program: CURSOR".

display "Enter your user id (default none): "
with no advancing.
accept userid.

if userid = spaces
EXEC SQL CONNECT TO sample END-EXEC
else
display "Enter your password : " with no advancing
accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
inspect passwd-name tallying passwd-length for characters
before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.
move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL DECLARE c1 CURSOR FOR
SELECT name, dept FROM staff
WHERE job='Mgr'
FOR UPDATE OF job END-EXEC.

1

EXEC SQL OPEN c1 END-EXEC.
move "OPEN CURSOR" to errloc.
call "checkerr" using SQLCA errloc.

2

```

perform Fetch-Loop thru End-Fetch-Loop
    until SQLCODE not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
move "CLOSE CURSOR" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL ROLLBACK END-EXEC.
move "ROLLBACK" to errloc.
call "checkerr" using SQLCA errloc.
DISPLAY "On second thought -- changes rolled back.".

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Main.
go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :PNAME, :DEPT END-EXEC.
if SQLCODE not equal 0
    go to End-Fetch-Loop.
display pname, " in dept. ", dept,
    " will be demoted to Clerk".
End-Fetch-Loop. exit.

End-Prog.
stop run.

```

4

3

FORTRAN Example: CURSOR.SQF

```
program cursor
implicit none
include 'sqlenv.f'
EXEC SQL INCLUDE SQLCA
EXEC SQL BEGIN DECLARE SECTION
  character*10  pname
  integer*2     dept
  character*8   userid
  character*18  passwd
EXEC SQL END DECLARE SECTION

character*80    errloc
print *, 'Sample FORTRAN program: CURSOR'
print *, 'Enter your user id (default none):'
read 101, userid
101 format (a8)
if( userid(1:1) .eq. ' ' ) then
  EXEC SQL CONNECT TO sample
else
  print *, 'Enter your password :'
  read 101, passwd
  EXEC SQL CONNECT TO sample USER :userid USING :passwd
end if
errloc = 'CONNECT'
call checkerr (sqlca, errloc, *999)

EXEC SQL DECLARE c1 CURSOR FOR 1
c      SELECT name, dept FROM staff WHERE job='Mgr'
c      FOR UPDATE OF job
EXEC SQL OPEN c1 2
errloc = 'OPEN'
call checkerr (sqlca, errloc, *999)

100 continue
EXEC SQL FETCH c1 INTO :pname, :dept 3
if (sqlcode .ne. 0) goto 200
print *, pname, ' in dept. ', dept, ' will be demoted to Clerk.'
goto 100

200 EXEC SQL CLOSE c1 4
errloc = 'CLOSE'
call checkerr (sqlca, errloc, *999)
EXEC SQL ROLLBACK
errloc = 'ROLLBACK'
call checkerr (sqlca, errloc, *999)
print *, 'On second thought -- changes rolled back.'
EXEC SQL CONNECT RESET
errloc = 'CONNECT RESET'
call checkerr (sqlca, errloc, *999)

999 stop
end
```

Updating and Deleting Retrieved Data

It is possible to update and delete the row referenced by a cursor. For a row to be updatable, the query corresponding to the cursor must not be read-only. For a description of what makes a query updatable or deletable, refer to the *SQL Reference*.

Updating Retrieved Data

To update with a cursor, use the WHERE CURRENT OF clause in an UPDATE statement. Use the FOR UPDATE clause to tell the system that you want to update some columns of the result table. You can specify a column in the FOR UPDATE without it being in the fullselect; therefore, you can update columns that are not explicitly retrieved by the cursor. If the FOR UPDATE clause is specified without column names, all columns of the table or view identified in the first FROM clause of the outer fullselect are considered to be updatable. Do not name more columns than you need in the FOR UPDATE clause. In some cases, naming extra columns in the FOR UPDATE clause can cause DB2 to be less efficient in accessing the data.

Deleting Retrieved Data

Deletion with a cursor is done using the WHERE CURRENT OF clause in a DELETE statement. In general, the FOR UPDATE clause is not required for deletion of the current row of a cursor. The only exception occurs when using dynamic SQL (see “Chapter 5. Writing Dynamic SQL Programs” on page 129 for information on dynamic SQL) for either the SELECT statement or the DELETE statement in an application which has been precompiled with LANGLEVEL set to SAA1, and bound with BLOCKING ALL. In this case, a FOR UPDATE clause is necessary in the SELECT statement. Refer to the *Command Reference* for information on the precompiler options.

The DELETE statement causes the row being referenced by the cursor to be deleted. This leaves the cursor positioned before the *next* row and a FETCH statement must be issued before additional WHERE CURRENT OF operations may be performed against the cursor.

Types of Cursors

Cursors fall into three categories:

Read only

The rows in the cursor can only be read, not updated. Read-only cursors are used when an application will only read data, not modify it. A cursor is considered read only if it is based on a read-only select-statement. See the rules in “Updating Retrieved Data” for select-statements which define non-updatable result tables.

There can be performance advantages for read-only cursors. For more information on read-only cursors, refer to the *Administration Guide, Performance*.

Updatable

The rows in the cursor can be updated. Updatable cursors are used when an application modifies data as the rows in the cursor are fetched. The specified query can only refer to one table or view. The query must also include the FOR UPDATE clause, naming each column that will be updated (unless the LANGLEVEL MIA precompile option is used).

Ambiguous

The cursor cannot be determined to be updatable or read only from its definition or context. This can happen when a dynamic SQL statement is encountered that could be used to change a cursor that would otherwise be considered read-only.

An ambiguous cursor is treated as read only if the BLOCKING ALL option is specified when precompiling or binding. Otherwise, it is considered updatable.

Note: Cursors processed dynamically are always ambiguous.

For a complete list of criteria used to determine whether a cursor is read-only, updatable, or ambiguous, refer to the *SQL Reference*.

Example: OPENFTCH Program

This example selects from a table using a cursor, opens the cursor, and fetches rows from the table. For each row fetched, it decides if the row should be deleted or updated (based on a simple criteria). The sample is available in the following programming languages:

C	openftch.sqc
Java	Openftch.sqlj and OpF_Curs.sqlj
COBOL	openftch.sqb
FORTTRAN	openftch.sqf

The REXX language does not support static SQL, so a sample is not provided.

How the OPENFTCH Program Works

1. **Declare the cursor.** The DECLARE CURSOR statement associates the cursor c1 to a query. The query identifies the rows that the application retrieves using the FETCH statement. The job field of staff is defined to be updatable, even though it is not specified in the result table.

2. **Open the cursor.** The cursor `c1` is opened, causing the database manager to perform the query and build a result table. The cursor is positioned *before* the first row.
3. **Retrieve a row.** The `FETCH` statement positions the cursor at the next row and moves the contents of the row into the host variables. This row becomes the *current* row.
4. **Update OR Delete the current row.** The current row is either updated or deleted, depending upon the value of `dept` returned with the `FETCH` statement.
 If an `UPDATE` is performed, the position of the cursor remains on this row because the `UPDATE` statement does not change the position of the current row.
 If a `DELETE` statement is performed, a different situation arises, because the *current* row is deleted. This is equivalent to being positioned before the *next* row, and a `FETCH` statement must be issued before additional `WHERE CURRENT OF` operations are performed.
5. **Close the cursor.** The `CLOSE` statement is issued, releasing the resources associated with the cursor. The cursor can be opened again, however.

The `CHECKERR` macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

- | | |
|-----------------|--|
| C | <code>check_error</code> is redefined as <code>CHECKERR</code> and is located in the <code>util.c</code> file. |
| COBOL | <code>CHECKERR</code> is an external program named <code>checkerr.cbl</code> . |
| FORTTRAN | <code>CHECKERR</code> is a subroutine located in the <code>util.f</code> file. |

See “Using `GET ERROR MESSAGE` in Example Programs” on page 120 for the source code for this error checking utility.

C Example: OPENFTCH.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
        char pname[10];
        short dept;
        char userid[9];
        char passwd[19];
    EXEC SQL END DECLARE SECTION;

#ifdef DB268K
    /* Before making any API calls for 68K environment,
       need to initial the Library Manager */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
#endif

    printf( "Sample C program: OPENFTCH\n" );

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: openftch [userid passwd]\n\n");
        return 1;
    } /* endif */

    EXEC SQL DECLARE c1 CURSOR FOR 1
        SELECT name, dept FROM staff WHERE job='Mgr'
        FOR UPDATE OF job;

    EXEC SQL OPEN c1; 2
    CHECKERR ("OPEN CURSOR");
```

```

do {
    EXEC SQL FETCH c1 INTO :pname, :dept; 3
    if (SQLCODE != 0) break;

    if (dept > 40) {
        printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
        EXEC SQL UPDATE staff SET job = 'Clerk' 4
            WHERE CURRENT OF c1;
        CHECKERR ("UPDATE STAFF");
    } else {
        printf( "%-10.10s in dept. %2d will be DELETED!\n",
            pname, dept);
        EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
        CHECKERR ("DELETE");
    } /* endif */
} while ( 1 );

EXEC SQL CLOSE c1; 5
CHECKERR ("CLOSE CURSOR");

EXEC SQL ROLLBACK;
CHECKERR ("ROLLBACK");
printf( "\nOn second thought -- changes rolled back.\n" );

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : OPENFTCH.SQC */

```

Java Example: Openftch.sqlj

OpF_Curs.sqlj

```
// PURPOSE : This file, named OpF_Curs.sqlj, contains the definition
//           of the class OpF_Curs used in the sample program Openftch.
```

```
import sqlj.runtime.ForUpdate;
#sql public iterator OpF_Curs implements ForUpdate (String, short);
```

Openftch.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Openftch
{   static
    {   try
        {   Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {   System.out.println ("\n  Error loading DB2 Driver...\n");
            System.out.println (e);
            System.exit(1);
        }
    }

    public static void main(String argv[])
    {   try
        {   System.out.println ("  Java Openftch Sample");
            String url = "jdbc:db2:sample";      // URL is jdbc:db2:dbname
            Connection con = null;

            // Set the connection
            if (argv.length == 0)
            {   // connect with default id/password
                con = DriverManager.getConnection(url);
            }
            else if (argv.length == 2)
            {   String userid = argv[0];
                String passwd = argv[1];
                // connect with user-provided username and password
                con = DriverManager.getConnection(url, userid, passwd);
            }
            else
            {   throw new Exception(
                "\nUsage: java Openftch [username password]\n");
            } // if - else if - else
            // Set the default context
            DefaultContext ctx = new DefaultContext(con);
            DefaultContext.setDefaultContext(ctx);
            // Enable transactions
            con.setAutoCommit(false);

            // Executing SQLJ positioned update/delete statements.
```

```

try
{
    OpF_Curs forUpdateCursor;
    String name = null;
    short dept=0;

    #sql forUpdateCursor =
    {
        SELECT name, dept
            FROM staff
            WHERE job='Mgr'
            FOR UPDATE OF job
    }; // #sql 1 2
    while (true)
    {
        #sql
        {
            FETCH :forUpdateCursor
                INTO :name, :dept
        }; // #sql 3
        if (forUpdateCursor.endFetch()) break;

        if (dept > 40)
        {
            System.out.println (
                name + " in dept. "
                + dept + " will be demoted to Clerk");
            #sql
            {
                UPDATE staff SET job = 'Clerk'
                WHERE CURRENT OF :forUpdateCursor
            }; // #sql 4
        }
        else
        {
            System.out.println (
                name + " in dept. " + dept
                + " will be DELETED!");
            #sql
            {
                DELETE FROM staff
                WHERE CURRENT OF :forUpdateCursor
            }; // #sql
        } // if - else
    }
    forUpdateCursor.close(); 5
}
catch( Exception e )
{
    throw e; }
finally
{
    // Rollback the transaction
    System.out.println("\nRollback the transaction...");
    #sql { ROLLBACK };
    System.out.println("Rollback done.");
} // try - catch - finally
}
catch( Exception e )
{
    System.out.println (e); } // try - catch
} // main
} // class Openftch

```

COBOL Example: OPENFTCH.SQB

Identification Division.
Program-ID. "openftch".

Data Division.
Working-Storage Section.

copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 pname pic x(10).
01 dept pic s9(4) comp-5.
01 userid pic x(8).
01 passwd.
49 passwd-length pic s9(4) comp-5 value 0.
49 passwd-name pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc pic x(80).

Procedure Division.
Main Section.

display "Sample COBOL program: OPENFTCH".

- * Get database connection information.
display "Enter your user id (default none): "
with no advancing.
accept userid.

if userid = spaces
EXEC SQL CONNECT TO sample END-EXEC
else
display "Enter your password : " with no advancing
accept passwd-name.
- * Passwords in a CONNECT statement must be entered in a VARCHAR format
- * with the length of the input string.
inspect passwd-name tallying passwd-length for characters
before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.
move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL DECLARE c1 CURSOR FOR
SELECT name, dept FROM staff
WHERE job='Mgr'
FOR UPDATE OF job END-EXEC.

1

EXEC SQL OPEN c1 END-EXEC
move "OPEN" to errloc.
call "checkerr" using SQLCA errloc.

2

- * call the FETCH and UPDATE/DELETE loop.

```

perform Fetch-Loop thru End-Fetch-Loop
until SQLCODE not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
move "CLOSE" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL ROLLBACK END-EXEC.
move "ROLLBACK" to errloc.
call "checkerr" using SQLCA errloc.
display "On second thought -- changes rolled back.".

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Main.
go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.
if SQLCODE not equal 0
go to End-Fetch-Loop.

if dept greater than 40
go to Update-Staff.

Delete-Staff.
display pname, " in dept. ", dept,
" will be DELETED!".

EXEC SQL DELETE FROM staff WHERE CURRENT OF c1 END-EXEC.
move "DELETE" to errloc.
call "checkerr" using SQLCA errloc.

go to End-Fetch-Loop.

Update-Staff.
display pname, " in dept. ", dept,
" will be demoted to Clerk".

EXEC SQL UPDATE staff SET job = 'Clerk'
WHERE CURRENT OF c1 END-EXEC.
move "UPDATE" to errloc.
call "checkerr" using SQLCA errloc.

End-Fetch-Loop. exit.

End-Prog.
stop run.

```

FORTRAN Example: OPENFTCH.SQF

```
program openftch
implicit none

include 'sqlenv.f'
EXEC SQL INCLUDE SQLCA
EXEC SQL BEGIN DECLARE SECTION
    character*10  pname
    integer*2     dept
    character*8   userid
    character*18  passwd
EXEC SQL END DECLARE SECTION

character*80      errloc

print *, 'Sample FORTRAN program: OPENFTCH'

print *, 'Enter your user id (default none):'
read 101, userid
101 format (a8)

if( userid(1:1) .eq. ' ' ) then
    EXEC SQL CONNECT TO sample
else
    print *, 'Enter your password :'
    read 101, passwd

    EXEC SQL CONNECT TO sample USER :userid USING :passwd
end if
errloc = 'CONNECT'
call checkerr (sqlca, errloc, *999)

EXEC SQL DECLARE c1 CURSOR FOR 1
c      SELECT name, dept FROM staff WHERE job='Mgr'
c      FOR UPDATE OF job

EXEC SQL OPEN c1 2
errloc = 'OPEN'
call checkerr (sqlca, errloc, *999)

100 continue
EXEC SQL FETCH c1 INTO :pname, :dept 3
if (sqlcode .ne. 0) goto 200

if (dept .gt. 40) then
    print *, pname, ' in dept. ', dept, ' will be demoted to Clerk.'
    EXEC SQL UPDATE staff SET job = 'Clerk' 4
c      WHERE CURRENT OF c1
    errloc = 'UPDATE'
    call checkerr (sqlca, errloc, *999)
endif

if (dept .lt. 41) then
    print *, pname, ' in dept. ', dept, ' will be DELETED!'
    EXEC SQL DELETE FROM staff WHERE CURRENT OF c1
```

```

        errloc = 'DELETE'
        call checkerr (sqlca, errloc, *999)
    endif
    goto 100

200 EXEC SQL CLOSE c1 5
    errloc = 'CLOSE'
    call checkerr (sqlca, errloc, *999)

EXEC SQL ROLLBACK
errloc = 'ROLLBACK'
call checkerr (sqlca, errloc, *999)
print *, 'On second thought -- changes rolled back.'

EXEC SQL CONNECT RESET
errloc = 'CONNECT RESET'
call checkerr (sqlca, errloc, *999)

999 stop
end

```

Advanced Scrolling Techniques

The following topics on advanced scrolling techniques are discussed in this section:

- Scrolling Through Data that has Already Been Retrieved
- Keeping a Copy of the Data
- Retrieving the Data a Second Time
- Establishing a Position at the End of a Table
- Updating Previously Retrieved Data

Scrolling Through Data that has Already Been Retrieved

When an application retrieves data from the database, the `FETCH` statement allows it to scroll forward through the data, however, the database manager has no embedded SQL statement that allows it scroll backwards through the data, (equivalent to a backward `FETCH`). DB2 CLI and Java, however, do support a backward `FETCH` through read-only scrollable cursors. Refer to the *CLI Guide and Reference* and see “Creating Java Applications and Applets” on page 538 for more information on scrollable cursors. For embedded SQL applications, you can use the following techniques to scroll through data that has been retrieved:

1. Keep a copy of the data that has been fetched and scroll through it by some programming technique.
2. Use SQL to retrieve the data again, typically by a second `SELECT` statement.

These options are discussed in more detail in:

- Keeping a Copy of the Data
- Retrieving the Data a Second Time

Keeping a Copy of the Data

An application can save fetched data in virtual storage. If the data does not fit in virtual storage, the application can write the data to a temporary file. One effect of this approach is that a user, scrolling backward, always sees exactly the same data that was fetched, even if the data in the database was changed in the interim by a transaction.

Using an isolation level of repeatable read, the data you retrieve from a transaction can be retrieved again by closing and opening a cursor. Other applications are prevented from updating the data in your result set. Isolation levels and locking can affect how users update data.

Retrieving the Data a Second Time

This technique depends on the order in which you want to see the data again:

- Retrieving from the Beginning
- Retrieving from the Middle
- Order of Rows in the Second Result Table
- Retrieving in Reverse Order

Retrieving from the Beginning

To retrieve the data again from the beginning, merely close the active cursor and reopen it. This action positions the cursor at the beginning of the result table. But, unless the application holds locks on the table, others may have changed it, so what had been the first row of the result table may no longer be.

Retrieving from the Middle

To retrieve data a second time from somewhere in the middle of the result table, execute a second SELECT statement and declare a second cursor on the statement. For example, suppose the first SELECT statement was:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
 ORDER BY DEPTNO
```

Now, suppose that you want to return to the rows that start with DEPTNO = 'M95' and fetch sequentially from that point. Code the following:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
 AND DEPTNO >= 'M95'
 ORDER BY DEPTNO
```

This statement positions the cursor where you want it.

Order of Rows in the Second Result Table

The rows of the second result table may not be displayed in the same order as in the first. The database manager does not consider the order of rows as significant unless the SELECT statement uses ORDER BY. Thus, if there are several rows with the same DEPTNO value, the second SELECT statement may retrieve them in a different order from the first. The only guarantee is that they will all be in order by department number, as demanded by the clause ORDER BY DEPTNO.

The difference in ordering could occur even if you were to execute the same SQL statement, with the same host variables, a second time. For example, the

statistics in the catalog could be updated between executions, or indexes could be created or dropped. You could then execute the SELECT statement again.

The ordering is more likely to change if the second SELECT has a predicate that the first did not have; the database manager could choose to use an index on the new predicate. For example, it could choose an index on LOCATION for the first statement in our example and an index on DEPTNO for the second. Because rows are fetched in order by the index key, the second order need not be the same as the first.

Again, executing two similar SELECT statements can produce a different ordering of rows, even if no statistics change and no indexes are created or dropped. In the example, if there are many different values of LOCATION, the database manager could choose an index on LOCATION for both statements. Yet changing the value of DEPTNO in the second statement to the following, could cause the database manager to choose an index on DEPTNO:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
     AND DEPTNO >= 'Z98'
     ORDER BY DEPTNO
```

Because of the subtle relationships between the form of an SQL statement and the values in this statement, never assume that two different SQL statements will return rows in the same order unless the order is uniquely determined by an ORDER BY clause.

Retrieving in Reverse Order

Ascending ordering of rows is the default. If there is only one row for each value of DEPTNO, then the following statement specifies a unique ascending ordering of rows:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
     ORDER BY DEPTNO
```

To retrieve the same rows in reverse order, specify that the order is descending, as in the following statement:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
     ORDER BY DEPTNO DESC
```

A cursor on the second statement retrieves rows in exactly the opposite order from a cursor on the first statement. Order of retrieval is guaranteed only if the first statement specifies a unique ordering sequence.

For retrieving rows in reverse order, it can be useful to have two indexes on the DEPTNO column, one in ascending order and the other in descending order.

Establishing a Position at the End of a Table

The database manager does not guarantee an order to data stored in a table; therefore, the end of a table is not defined. However, order is defined on the result of an SQL statement:

```
SELECT * FROM DEPARTMENT
ORDER BY DEPTNO DESC
```

For this example, the following statement positions the cursor at the row with the highest DEPTNO value:

```
SELECT * FROM DEPARTMENT
WHERE DEPTNO =
(SELECT MAX(DEPTNO) FROM DEPARTMENT)
```

Note, however, that if several rows have the same value, the cursor is positioned on the first of them.

Updating Previously Retrieved Data

To scroll backward and update data that was retrieved previously, you can use a combination of the techniques discussed in “Scrolling Through Data that has Already Been Retrieved” on page 101 and “Updating Retrieved Data” on page 90. You can do one of two things:

1. If you have a second cursor on the data to be updated and if the SELECT statement uses none of the restricted elements, you can use a cursor-controlled UPDATE statement. Name the second cursor in the WHERE CURRENT OF clause.
2. In other cases, use UPDATE with a WHERE clause that names all the values in the row or specifies the primary key of the table. You can execute one statement many times with different values of the variables.

Example: UPDAT Program

The UPDAT program uses dynamic SQL to access the STAFF table in the SAMPLE database and changes all managers to clerks. Then the program reverses the changes by rolling back the unit of work. The sample is available in the following programming languages:

C	updat.sqc
Java	Updat.sqlj
COBOL	updat.sqb
FORTRAN	updat.sqf

How the UPDAT Program Works

1. **Define an SQLCA structure.** The INCLUDE SQLCA statement defines and declares an SQLCA structure, and defines SQLCODE as an element within the structure. The SQLCODE field of the SQLCA structure is updated with error information by the database manager after execution of SQL statements and database manager API calls.

Java applications access SQLCODE and SQLSTATE through the methods defined for the SQLException object, and therefore do not need an equivalent "include SQLCA" statement.

REXX applications have one occurrence of an SQLCA structure, named SQLCA, predefined for application use. It can be referenced without application definition.

2. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are used to pass data to and from the database manager. They are prefixed with a colon (:) when referenced in an SQL statement. Java and REXX applications do not need to declare host variables, except (for REXX) in the case of LOB file reference variables and locators. Host variable data types and sizes are determined at run time when the variables are referenced.
3. **Connect to database.** The program connects to the sample database, and requests shared access to it. (It is assumed that a START DATABASE MANAGER API call or db2start command has been issued.) Other programs that connect to the same database using shared access are also granted access.
4. **Execute the UPDATE SQL statement.** The SQL statement is executed *statically* with the use of a host variable. The job column of the staff tables is set to the value of the host variable, where the job column has a value of Mgr.
5. **Execute the DELETE SQL statement** The SQL statement is executed *statically* with the use of a host variable. All rows that have a job column value equal to that of the specified host variable, (jobUpdate/job-update/job_update) are deleted.
6. **Execute the INSERT SQL statement** A row is inserted into the STAFF table. This insertion implements the use of a host variable which was set prior to the execution of this SQL statement.
7. **End the transaction.** End the *unit of work* with a ROLLBACK statement. The result of the SQL statement executed previously can be either made permanent using the COMMIT statement, or undone using the ROLLBACK statement. All SQL statements within the *unit of work* are affected.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C	check_error is redefined as CHECKERR and is located in the util.c file.
Java	Any SQL error is thrown as an SQLException and handled in the catch block of the application.
COBOL	CHECKERR is an external program named checkerr.cbl.
FORTTRAN	CHECKERR is a subroutine located in the util.f file.
REXX	CHECKERR is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

C Example: UPDAT.SQC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlenv.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA; 1

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION; 2
        char statement[256];
        char userid[9];
        char passwd[19];
        char jobUpdate[6];
    EXEC SQL END DECLARE SECTION;

#ifdef DB268K
    /* Before making any API calls for 68K environment,
       need to initial the Library Manager */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
#endif

    printf( "\nSample C program:  UPDAT \n");

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd; 3
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: updat [userid passwd]\n\n");
        return 1;
    } /* endif */

    strcpy (jobUpdate, "Clerk");
    EXEC SQL UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr'; 4
    CHECKERR ("UPDATE STAFF");
    printf ("All 'Mgr' have been demoted to 'Clerk'!\n" );

    strcpy (jobUpdate, "Sales");
```

```

EXEC SQL DELETE FROM staff WHERE job = :jobUpdate; 5
CHECKERR ("DELETE FROM STAFF");
printf ("All 'Sales' people have been deleted!\n");

EXEC SQL INSERT INTO staff
VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0); 6
CHECKERR ("INSERT INTO STAFF");
printf ("New data has been inserted\n");

EXEC SQL ROLLBACK; 7
CHECKERR ("ROLLBACK");
printf( "On second thought -- changes rolled back.\n" );

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : UPDAT.SQC */

```


Java Example: Updat.sqlj

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

class Updat
{
    static
    {
        try
        {
            Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {
            System.out.println ("\n Error loading DB2 Driver...\n");
            System.out.println (e);
            System.exit(1);
        }
    }
}

public static void main(String argv[])
{
    try
    {
        System.out.println ("\n Java Updat Sample");

        String url = "jdbc:db2:sample";          // URL is jdbc:db2:dbname
        Connection con = null;

        // Set the connection 3
        if (argv.length == 0)
        { // connect with default id/password
            con = DriverManager.getConnection(url);
        }
        else if (argv.length == 2)
        {
            String userid = argv[0];
            String passwd = argv[1];

            // connect with user-provided username and password
            con = DriverManager.getConnection(url, userid, passwd);
        }
        else
        {
            throw new Exception("\nUsage: java Updat [username password]\n");
        }

        // Set the default context
        DefaultContext ctx = new DefaultContext(con);
        DefaultContext.setDefaultContext(ctx);

        // Enable transactions
        con.setAutoCommit(false);

        // UPDATE/DELETE/INSERT
        try
        {
            String jobUpdate = null;
            jobUpdate="Clerk";
            #sql {UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr'}; 4
            System.out.println("\nAll 'Mgr' have been demoted to 'Clerk'!");
        }
    }
}
```

```

        jobUpdate="Sales";
        #sql {DELETE FROM staff WHERE job = :jobUpdate};
        System.out.println("All 'Sales' people have been deleted!"); 5

        #sql {INSERT INTO staff VALUES (999,'Testing',99,:jobUpdate,0,0,0)}; 6
        System.out.println("New data has been inserted");
    }
    catch( Exception e )
    {
        throw e;
    }
    finally
    {
        // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        #sql { ROLLBACK }; 7
        System.out.println("Rollback done.");
    }
}
catch (Exception e)
{
    System.out.println (e);
}
}

```

COBOL Example: UPDAT.SQB

Identification Division.
Program-ID. "updat".

Data Division.
Working-Storage Section.

copy "sql.cbl".
copy "sqlenv.cbl".
copy "sqlca.cbl".

1

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

2

01 statement pic x(80).
01 userid pic x(8).
01 passwd.
 49 passwd-length pic s9(4) comp-5 value 0.
 49 passwd-name pic x(18).
01 job-update pic x(5).
EXEC SQL END DECLARE SECTION END-EXEC.

* Local variables

77 errloc pic x(80).
77 error-rc pic s9(9) comp-5.
77 state-rc pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API

* Use application specific bound instead of BUFFER-SZ

77 buffer-size pic s9(4) comp-5 value 1024.
77 line-width pic s9(4) comp-5 value 80.
77 error-buffer pic x(1024).
77 state-buffer pic x(1024).

Procedure Division.

Main Section.

display "Sample COBOL program: UPDAT".

display "Enter your user id (default none): "
 with no advancing.
accept userid.

if userid = spaces
 EXEC SQL CONNECT TO sample END-EXEC
else
 display "Enter your password : " with no advancing
 accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.

inspect passwd-name tallying passwd-length for characters
 before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.

3

move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

```

move "Clerk" to job-update.
EXEC SQL UPDATE staff SET job=:job-update
      WHERE job='Mgr' END-EXEC.
move "UPDATE STAFF" to errloc.
call "checkerr" using SQLCA errloc.

display "All 'Mgr' have been demoted to 'Clerk'!".

move "Sales" to job-update.
EXEC SQL DELETE FROM staff WHERE job=:job-update END-EXEC.
move "DELETE FROM STAFF" to errloc.
call "checkerr" using SQLCA errloc.

display "All 'Sales' people have been deleted!".

EXEC SQL INSERT INTO staff VALUES (999, 'Testing', 99,
      :job-update, 0, 0, 0) END-EXEC.
move "INSERT INTO STAFF" to errloc.
call "checkerr" using SQLCA errloc.

display "New data has been inserted".

EXEC SQL ROLLBACK END-EXEC.
move "ROLLBACK" to errloc.
call "checkerr" using SQLCA errloc.

DISPLAY "On second thought -- changes rolled back."

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.

End-Prog.
stop run.

```

FORTRAN Example: UPDAT.SQF

```
program updat
implicit none

include 'sql.f'
include 'sqlenv.f'
EXEC SQL INCLUDE SQLCA 1

EXEC SQL BEGIN DECLARE SECTION 2
  character*80  statement
  character*8   userid
  character*18  passwd
  character*5   job_update
EXEC SQL END DECLARE SECTION

character*80    errloc

print *, 'Sample FORTRAN program: UPDAT'

print *, 'Enter your user id (default none):'
read 100, userid
100 format (a8)

if( userid(1:1) .eq. ' ' ) then
  EXEC SQL CONNECT TO sample
else
  print *, 'Enter your password :'
  read 100, passwd

  EXEC SQL CONNECT TO sample USER :userid USING :passwd
end if 3
errloc = 'CONNECT'
call checkerr (sqlca, errloc, *999)

job_update='Clerk'
EXEC SQL UPDATE staff SET job = :job_update
+ WHERE job = 'Mgr' 4
errloc = 'UPDATE STAFF'
call checkerr (sqlca, errloc, *999)
print *, 'All 'Mgr' have been demoted to 'Clerk!''

job_update='Sales'
EXEC SQL DELETE FROM STAFF WHERE job = :job_update 5
errloc = 'DELETE FROM STAFF'
call checkerr (sqlca, errloc, *999)
print *, 'All 'Sales' people have been deleted!'

EXEC SQL INSERT INTO staff VALUES (999, 'Testing', 99, 6
+ :job_update, 0, 0, 0)
errloc = 'INSERT INTO STAFF'
call checkerr (sqlca, errloc, *999)
print *, 'New data has been inserted'

EXEC SQL ROLLBACK 7
errloc = 'ROLLBACK'
```

```
call checkerr (sqlca, errloc, *999)
print *, 'On second thought -- changes rolled back.'

EXEC SQL CONNECT RESET
errloc = 'CONNECT RESET'
call checkerr (sqlca, errloc, *999)

999 stop
end
```

REXX Example: UPDAT.CMD

Note: REXX programs cannot contain static SQL. This program is written with dynamic SQL.

```
/* this variable (SYSTEM) must be user defined */
SYSTEM = AIX
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS', 'DB2AR', 'SQLDBS' )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("db2rexx")

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
PARSE ARG dbname userid password .
if ((dbname = "" ) | ,
    (userid <> "" & password = "" ) ,
    ) then do
  SAY "USAGE: updat.cmd <dbname> [<userid> <password>]"

  exit -1
end

/* connect to database */
SAY
SAY 'Connect to' dbname
IF password= "" THEN
  CALL SQLEXEC 'CONNECT TO' dbname
ELSE
  CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password 3

CALL CHECKERR 'Connect to '
SAY "Connected"

say 'Sample REXX program: UPDAT.CMD'

jobupdate = "'Clerk'"
st = "UPDATE staff SET job =" jobupdate "WHERE job = 'Mgr'"
call SQLEXEC 'EXECUTE IMMEDIATE :st' 4
call CHECKERR 'UPDATE'
say "All 'Mgr' have been demoted to 'Clerk'!"

jobupdate = "'Sales'"
st = "DELETE FROM staff WHERE job =" jobupdate
call SQLEXEC 'EXECUTE IMMEDIATE :st' 5
call CHECKERR 'DELETE'
say "All 'Sales' people have been deleted!"
```

```

st = "INSERT INTO staff VALUES (999, 'Testing', 99, ' jobupdate ", 0, 0, 0)"
call SQLEXEC 'EXECUTE IMMEDIATE :st' 6
call CHECKERR 'INSERT'
say 'New data has been inserted'

call SQLEXEC 'ROLLBACK' 7
call CHECKERR 'ROLLBACK'
say 'On second thought...changes rolled back.'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'

CHECKERR:
  arg errloc

  if ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /*****\
    * GET ERROR MESSAGE API called *
    \*****/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
end
return 0

```

Diagnostic Handling and the SQLCA Structure

Applications issuing SQL statements and calling database manager APIs must properly check for error conditions by examining return codes and the SQLCA structure.

Return Codes

Most database manager APIs pass back a zero return code when successful. In general, a non-zero return code indicates that the secondary error handling mechanism, the SQLCA structure, may be corrupt. In this case, the called API is not executed. A possible cause for a corrupt SQLCA structure is passing an invalid address for the structure.

SQLCODE and SQLSTATE

Error information is returned in the SQLCODE and SQLSTATE fields of the SQLCA structure, which is updated after every executable SQL statement and most database manager API calls.

A source file containing executable SQL statements can provide at least one SQLCA structure with the name `sqlca`. The SQLCA structure is defined in the SQLCA include file. Source files without embedded SQL statements, but calling database manager APIs, can also provide one or more SQLCA structures, but their names are arbitrary.

If your application is compliant with the FIPS 127-2 standard, you can declare the SQLSTATE and SQLCODE as host variables instead of using the SQLCA structure. For information on how to do this, see “SQLSTATE and SQLCODE Variables” on page 530 for C or C++ applications, “SQLSTATE and SQLCODE Variables” on page 589 for COBOL applications, or “SQLSTATE and SQLCODE Variables” on page 604 for FORTRAN applications.

An SQLCODE value of 0 means successful execution (with possible SQLWARN warning conditions). A positive value means that the statement was successfully executed but with a warning, as with truncation of a host variable. A negative value means that an error condition occurred.

An additional field, SQLSTATE, contains a standardized error code consistent across other IBM database products and across SQL92 conformant database managers. Practically speaking, you should use SQLSTATEs when you are concerned about portability since SQLSTATEs are common across many database managers.

The SQLWARN field contains an array of warning indicators, even if SQLCODE is zero. The first element of the SQLWARN array, SQLWARN0,

contains a blank if all other elements are blank. SQLWARN0 contains a W if at least one other element contains a warning character.

Refer to the *Administrative API Reference* for more information about the SQLCA structure, and the *Message Reference* for a listing of SQLCODE and SQLSTATE error conditions.

Note: If you want to develop applications that access various IBM RDBMS servers you should:

- Where possible, have your applications check the SQLSTATE rather than the SQLCODE.
- If your applications will use DB2 Connect, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.

Token Truncation in SQLCA Structure

Since tokens may be truncated in the SQLCA structure, you should not use the token info for diagnostic purposes. While you can define table and column names with lengths of up to 128 bytes, the SQLCA tokens will be truncated to 17 bytes plus a truncation terminator (>). Application logic should not depend on actual values of the sqlerrmc field. Refer to the *SQL Reference* for a description of the SQLCA structure, and a discussion of token truncation.

Handling Errors using the WHENEVER Statement

The WHENEVER statement causes the precompiler to generate source code that directs the application to go to a specified label if an error, warning, or if no rows are found during execution. The WHENEVER statement affects all subsequent executable SQL statements until another WHENEVER statement alters the situation.

The WHENEVER statement has three basic forms:

```
EXEC SQL WHENEVER SQLERROR   action
EXEC SQL WHENEVER SQLWARNING action
EXEC SQL WHENEVER NOT FOUND  action
```

In the above statements:

SQLERROR

Identifies any condition where SQLCODE < 0.

SQLWARNING

Identifies any condition where SQLWARN(0) = W or SQLCODE > 0 but is not equal to 100.

NOT FOUND

Identifies any condition where `SQLCODE = 100`.

In each case, the *action* can be either of the following:

CONTINUE

Indicates to continue with the next instruction in the application.

GO TO *label*

Indicates to go to the statement immediately following the label specified after GO TO. (GO TO can be two words, or one word, GOTO.)

If the WHENEVER statement is not used, the default action is to continue processing if an error, warning, or exception condition occurs during execution.

The WHENEVER statement must appear before the SQL statements you want to affect. Otherwise, the precompiler does not know that additional error-handling code should be generated for the executable SQL statements. You can have any combination of the three basic forms active at any time. The order in which you declare the three forms is not significant. To avoid an infinite looping situation, ensure that you undo the WHENEVER handling before any SQL statements are executed inside the handler. You can do this using the WHENEVER CONTINUE statement.

For a complete description of the WHENEVER statement, refer to the *SQL Reference*.

Exception, Signal, Interrupt Handler Considerations

An exception, signal, or interrupt handler is a routine that gets control when an exception, signal, or interrupt occurs. The type of handler applicable is determined by your operating environment, as shown in the following:

Windows 32-bit Operating Systems

Pressing Ctrl-C or Ctrl-Break generates an interrupt.

OS/2 Pressing Ctrl-C or Ctrl-Break generates an operating system exception.

UNIX Usually, pressing Ctrl-C generates the SIGINT interrupt signal. Note that keyboards can easily be redefined so SIGINT may be generated by a different key sequence on your machine.

For other operating systems that are not in the above list, refer to the *Application Building Guide*.

Do not put SQL statements (other than COMMIT or ROLLBACK) in exception, signal, and interrupt handlers. With these kinds of error conditions, you normally want to do a ROLLBACK to avoid the risk of inconsistent data.

Note that you should exercise caution when coding a COMMIT and ROLLBACK in exception/signal/interrupt handlers. If you call either of these statements by themselves, the COMMIT or ROLLBACK is not executed until the current SQL statement is complete, if one is running. This is not the behavior desired from a Ctrl-C handler.

The solution is to call the INTERRUPT API (sqleintr/sqlgintr) before issuing a ROLLBACK. This interrupts the current SQL query (if the application is executing one) and lets the ROLLBACK begin immediately. If you are going to perform a COMMIT rather than a ROLLBACK, you do not want to interrupt the current command.

When using APPC to access a remote database server (DB2 for AIX or host database system using DB2 Connect), the application may receive a SIGUSR1 signal. This signal is generated by SNA Services/6000 when an unrecoverable error occurs and the SNA connection is stopped. You may want to install a signal handler in your application to handle SIGUSR1.

Refer to your platform documentation for specific details on the various handler considerations.

Exit List Routine Considerations

Do not use SQL or DB2 API calls in exit list routines. Note that you cannot disconnect from a database in an exit routine.

Using GET ERROR MESSAGE in Example Programs

The code clips shown in “C Example: UTIL.C”, “COBOL Example: CHECKERR.CBL” on page 124, and “FORTRAN Example: UTIL.F” on page 126, demonstrate the use of the GET ERROR MESSAGE API to obtain the corresponding information related to the SQLCA passed in.

You can find information on building these examples in the README files, or in the header section of these sample programs.

C Example: UTIL.C

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlenv.h>
#include <sqlda.h>
#include <sqlca.h>
#include <string.h>
```

```

#include <ctype.h>

/* #include "system.h" */

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

#define SQLSTATE sqlca.sqlstate
#ifdef max
#define max(A, B) ((A) > (B) ? (A) : (B))
#endif
#ifdef min
#define min(A, B) ((A) > (B) ? (B) : (A))
#endif
#define CHECKERR(CE_STR)      check_error (CE_STR, &sqlca)
#define LOBLENGTH 29
#ifdef DB2OS2
#include <conio.h>
#elif defined (DB2AIX)
#include <curses.h>
#endif

short   *errdata = NULL;      /* Hold error counts */

#define mem_error(xx) \
{ printf("SQCERROR--Out of memory when %s.\n",xx); \
  exit(-1); }
#define MAXERRORS 65536

struct lob_file {
    unsigned long name_length;
    unsigned long data_length;
    unsigned long file_option;
    char          name[255];
};

struct col_info {
    char  name[19]; /* name of column */
    int   coltype; /* column type */
    short length; /* length of the column */
    short colno; /* column number */
};

struct tab_info {
    char  name[19], /* name of table */
        creator[9], /* creator of the table */
        type;
    short colcount; /* number of columns */
    struct col_info **alpha_list, /* alphabetical listing of columns */
        **num_list; /* numerical listing of columns */
};

/*****

```

```

** Procedure : check_error
**
** Purpose : This procedure checks the SQLCACODE flag and prints out any
**            information that is available related to the specific error.
**
**
*****
int check_error (char eString[], struct sqlca *caPointer) {
    char eBuffer[1024];
    char sBuffer[1024];
    short rc, Erc;

    if (caPointer->sqlcode != 0) {
        printf ("--- error report ---\n");
        printf ("ERROR occurred : %s.\nSQLCODE : %ld\n", eString,
            caPointer->sqlcode);

        /*****\
        * GET SQLSTATE MESSAGE *
        \*****/
        rc = sqlgostt (sBuffer, 1024, 80, caPointer->sqlstate);

        /*****\
        * GET ERROR MESSAGE API called *
        \*****/
        Erc = sqlaintp (eBuffer, 1024, 80, caPointer);

        /* return code is the length of the eBuffer string */
        if (Erc > 0) printf ("%s", eBuffer);

        if (caPointer->sqlcode < 0) {
            if (rc == 0) {
                printf ("\n%s", sBuffer);
            }

            printf ("--- end error report ---\n");
            return 1;
        } else {
            /* errorCode is just a Warning message */
            if (rc == 0) {
                printf ("\n%s", sBuffer);
            }

            printf ("--- end error report ---\n");
            printf ("WARNING - CONTINUING PROGRAM WITH WARNINGS!\n");
            return 0;
        } /* endif */
    } /* endif */
    return 0;
}

.
.
.
/* functions / procedures on setting up and outputting info from SQLDA */
.
.
.

```

Java Example: Catching SQLException

JDBC and SQLJ applications throw an SQLException when an error occurs during SQL processing. Your applications can catch and display an SQLException with the following code:

```
try {
    Statement stmt = connection.createStatement();
    int rowsDeleted = stmt.executeUpdate(
        "DELETE FROM employee WHERE empno = '000010'");
    System.out.println( rowsDeleted + " rows were deleted");
}

catch (SQLException sqle) {
    System.out.println(sqle);
}
```

For more information on handling SQLExceptions, see “SQLSTATE and SQLCODE Values” on page 537.

COBOL Example: CHECKERR.CBL

```
Identification Division.
Program-ID. "checkerr".

Data Division.
Working-Storage Section.

copy "sql.cbl".

* Local variables
77 error-rc          pic s9(9) comp-5.
77 state-rc          pic s9(9) comp-5.

* Variables for the GET ERROR MESSAGE API
* Use application specific bound instead of BUFFER-SZ
* 77 buffer-size      pic s9(4) comp-5 value BUFFER-SZ.
* 77 error-buffer      pic x(BUFFER-SZ).
* 77 state-buffer      pic x(BUFFER-SZ).
77 buffer-size        pic s9(4) comp-5 value 1024.
77 line-width         pic s9(4) comp-5 value 80.
77 error-buffer        pic x(1024).
77 state-buffer        pic x(1024).

Linkage Section.
copy "sqlca.cbl" replacing ==VALUE "SQLCA"    "==" by == ==
                        ==VALUE 136==        by == ==.
01 errloc              pic x(80).

Procedure Division using sqlca errloc.
Checkerr Section.
    if SQLCODE equal 0
        go to End-Checkerr.

    display "--- error report ---".
    display "ERROR occurred : ", errloc.
    display "SQLCODE : ", SQLCODE.

*****
* GET ERROR MESSAGE API called *
*****
    call "sqlgintp" using
        by value      buffer-size
        by value      line-width
        by reference sqlca
        by reference error-buffer
        returning error-rc.

*****
* GET SQLSTATE MESSAGE *
*****
    call "sqlgstt" using
        by value      buffer-size
        by value      line-width
        by reference sqlstate
        by reference state-buffer
```



```

        returning state-rc.

if error-rc is greater than 0
    display error-buffer.

if state-rc is greater than 0
    display state-buffer.

if state-rc is less than 0
    display "return code from GET SQLSTATE =" state-rc.

if SQLCODE is less than 0
    display "--- end error report ---"
    go to End-Prog.

display "--- end error report ---"
display "CONTINUING PROGRAM WITH WARNINGS!".
End-Checkerr. exit program.
End-Prog. stop run.

```

FORTRAN Example: UTIL.F

```
subroutine checkerr (ca, errloc, *)
character      ca(136)
character*60   errloc

include 'sqlenv.f'
include 'sqlutil.f'
include 'sqlca_cn.f'
include 'sql.f'
integer*4      error_rc
integer*4      state_rc
integer*2      line_width

*      Use application-specific bound instead of INTBUFSZ
*      integer*2      buffer_size /INTBUFSZ/
*      character      error_buffer(INTBUFSZ)
*      character      state_buffer(INTBUFSZ)
integer*2      buffer_size /1024/
character      error_buffer(1024)
character      state_buffer(1024)

error_rc = sqlgmcpy (%ref(sqlca),
+                  %ref(ca),
+                  %val(136))

if ( sqlcode .ne. 0 ) then
  print *, '--- error report ---'
  print *, 'ERROR occurred : ', errloc

*****
*      GET ERROR MESSAGE *
*****
      error_rc = sqlgintp ( %val(buffer_size),
+                          %val(line_width),
+                          %ref(sqlca),
+                          %ref(error_buffer))

*****
*      GET SQLSTATE MESSAGE *
*****
      state_rc = sqlggstt (%val(buffer_size),
+                          %val(line_width),
+                          %ref(sqlstate),
+                          %ref(state_buffer))
      if (state_rc .gt. 0) then
        print *, (state_buffer(i), i=1,state_rc)
      endif

      if (state_rc .lt. 0) then
        print *, 'return code from GET SQLSTATE = ', state_rc
      endif

      print *, 'SQLCODE : ', sqlcode
```

```

if (error_rc .gt. 0) then
  print *, (error_buffer(i), i=1,error_rc)
  print *, '--- end error report ---'
  if (sqlcode .lt. 0) then
    return 1
  endif

  if (sqlcode .gt. 0) then
    print *, 'CONTINUING PROGRAM WITH WARNINGS'
    return
  endif
endif

print *, 'DID NOT GET ERROR MESSAGE'
return 1
endif
return
end

```

REXX Example: CHECKERR Procedure

```
/* this variable (SYSTEM) must be user defined */
SYSTEM = AIX
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS', 'DB2AR', 'SQLDBS' )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("db2rex")
  .
  .
  .
call CHECKERR 'INSERT'
  .
  .
  .
CHECKERR:
  arg errloc

  if ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /*****\
    * GET ERROR MESSAGE API called *
    \*****/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
end
return 0
```

Chapter 5. Writing Dynamic SQL Programs

Why Use Dynamic SQL?	129	Allocating an SQLDA Structure	152
Dynamic SQL Support Statements	129	Passing Data Using an SQLDA Structure	156
Comparing Dynamic SQL with Static SQL	130	Processing Interactive SQL Statements	157
Using PREPARE, DESCRIBE, FETCH and the SQLDA.	133	Determining Statement Type	157
Declaring and Using Cursors.	134	Varying-List SELECT Statement	158
Example: Dynamic SQL Program	136	Saving SQL Requests from End Users	158
How the Dynamic Program Works	136	Example: ADHOC Program	159
C Example: DYNAMIC.SQC	138	How the ADHOC Program Works	159
Java Example: Dynamic.java	140	C Example: ADHOC.SQC	162
COBOL Example: DYNAMIC.SQB	142	Variable Input to Dynamic SQL	166
FORTRAN Example: DYNAMIC.SQF	144	Using Parameter Markers	166
REXX Example: DYNAMIC.CMD	146	Example: VARINP Program	167
Declaring the SQLDA	148	How the VARINP Program Works	167
Preparing the Statement Using the Minimum SQLDA Structure	149	C Example: VARINP.SQC	169
Allocating an SQLDA with Sufficient SQLVAR Entries	150	Java Example: Varinp.java.	171
Describing the SELECT Statement	151	COBOL Example: VARINP.SQB	173
Acquiring Storage to Hold a Row	151	FORTRAN Example: VARINP.SQF	175
Processing the Cursor	152	The DB2 Call Level Interface (CLI)	177
		Comparing Embedded SQL and DB2 CLI	177
		Advantages of Using DB2 CLI	178
		Deciding on Embedded SQL or DB2 CLI	180

Why Use Dynamic SQL?

You may want to use dynamic SQL when:

- You need all or part of the SQL statement to be generated during application execution.
- The objects referenced by the SQL statement do not exist at precompile time.
- You want the statement to always use the most optimal access path, based on current database statistics.
- You want to modify the compilation environment of the statement, that is, experiment with the special registers.

Dynamic SQL Support Statements

The dynamic SQL support statements accept a character-string host variable and a statement name as arguments. The host variable contains the SQL statement to be processed dynamically in text form. The statement text is not processed when an application is precompiled. In fact, the statement text does not have to exist at the time the application is precompiled. Instead, the SQL

statement is treated as a host variable for precompilation purposes and the variable is referenced during application execution. These SQL statements are referred to as *dynamic SQL*.

Dynamic SQL support statements are required to transform the host variable containing SQL text into an executable form and operate on it by referencing the statement name. These statements are:

EXECUTE IMMEDIATE

Prepares and executes a statement that does not use any host variables. All EXECUTE IMMEDIATE statements in an application are cached in the same place at run time, so only the last statement is known. Use this statement as an alternative to the PREPARE and EXECUTE statements.

PREPARE

Turns the character string form of the SQL statement into an executable form of the statement, assigns a statement name, and optionally places information about the statement in an SQLDA structure.

EXECUTE

Executes a previously prepared SQL statement. The statement can be executed repeatedly within a connection.

DESCRIBE

Places information about a prepared statement into an SQLDA.

An application can execute most SQL statements dynamically. See Table 26 on page 629 for the complete list of supported SQL statements.

Note: The content of dynamic SQL statements follows the same syntax as static SQL statements, but with the following exceptions:

- Comments are not allowed.
- The statement cannot begin with EXEC SQL.
- The statement cannot end with the statement terminator. An exception to this is the CREATE TRIGGER statement which can contain a semicolon (;).

Comparing Dynamic SQL with Static SQL

The question of whether to use static or dynamic SQL for performance is usually of great interest to programmers. The answer, of course, is that it all depends on your situation. Refer to Table 6 on page 131 to help you decide whether to use static or dynamic SQL. There may be certain considerations such as security which dictate static SQL, or your environment (such as whether you are using DB2 CLI or the CLP) which dictates dynamic SQL.

When making your decision, consider the following recommendations on whether to choose static or dynamic SQL in a particular situation. In the following table, 'either' means that there is no advantage to either static or dynamic SQL. Note that these are general recommendations only. Your specific application, its intended usage, and working environment dictate the actual choice. When in doubt, prototyping your statements as static SQL, then as dynamic SQL, and then comparing the differences is the best approach.

Table 6. Comparing Static and Dynamic SQL

Consideration	Likely Best Choice
Time to run the SQL statement:	
• Less than 2 seconds	• Static
• 2 to 10 seconds	• either
• More than 10 seconds	• Dynamic
Data Uniformity	
• Uniform data distribution	• Static
• Slight non-uniformity	• either
• Highly non-uniform distribution	• Dynamic
Range (<,>,BETWEEN,LIKE) Predicates	
• Very Infrequent	• Static
• Occasional	• either
• Frequent	• Dynamic
Repetitious Execution	
• Runs many times (10 or more times)	• either
• Runs a few times (less than 10 times)	• either
• Runs once	• Static
Nature of Query	
• Random	• Dynamic
• Permanent	• either
Run Time Environment (DML/DDL)	
• Transaction Processing (DML Only)	• either
• Mixed (DML and DDL - DDL affects packages)	• Dynamic
• Mixed (DML and DDL - DDL does not affect packages)	• either
Frequency of RUNSTATS	
• Very infrequently	• Static
• Regularly	• either
• Frequently	• Dynamic

In general, an application using dynamic SQL has a higher start-up (or initial) cost per SQL statement due to the need to compile the SQL statements prior to using them. Once compiled, the execution time for dynamic SQL compared to static SQL should be equivalent and, in some cases, faster due to better access plans being chosen by the optimizer. Each time a dynamic statement is executed, the initial compilation cost becomes less of a factor. If multiple users are running the same dynamic application with the same statements, only the first application to issue the statement realizes the cost of statement compilation.

In a mixed DML and DDL environment, the compilation cost for a dynamic SQL statement may vary as the statement may be implicitly recompiled by the system while the application is running. In a mixed environment, the choice between static and dynamic SQL must also factor in the frequency in which packages are invalidated. If the DDL does invalidate packages, dynamic SQL may be more efficient as only those queries executed are recompiled when they are next used. Others are not recompiled. For static SQL, the entire package is rebound once it has been invalidated.

Now suppose your particular application contains a mixture of the above characteristics and some of these characteristics suggest that you use static while others suggest dynamic. In this case, there is no clear cut decision and you should probably use whichever method you have the most experience with, and with which you feel most comfortable. Note that the considerations in the above table are listed roughly in order of importance.

Note: Static and dynamic SQL each come in two types that make a difference to the DB2 optimizer. These are:

1. Static SQL containing no host variables

This is an unlikely situation which you may see only for:

- *Initialization* code
- Novice training examples

This is actually the best combination from a performance perspective in that there is no run-time performance overhead and yet the DB2 optimizer's capabilities can be fully realized.

2. Static SQL containing host variables

This is the traditional *legacy* style of DB2 applications. It avoids the run time overhead of a PREPARE and catalog locks acquired during statement compilation. Unfortunately, the full power of the optimizer cannot be harnessed since it does not know the entire SQL statement. A particular problem exists with highly non-uniform data distributions.

3. Dynamic SQL containing no parameter markers

This is the typical style for random query interfaces (such as the CLP) and is the optimizer's preferred flavor of SQL. For complex queries, the overhead of the PREPARE statement is usually worthwhile due to improved execution time. For more information on parameter markers, see "Using Parameter Markers" on page 166.

4. Dynamic SQL containing parameter markers

This is the most common type of SQL for CLI applications. The key benefit is that the presence of parameter markers allows the cost of the PREPARE to be amortized over the repeated executions of the statement, typically a select or insert. This amortization is true for all repetitive dynamic SQL applications. Unfortunately, just like static SQL with host variables, parts of the DB2 optimizer will not work since complete information is unavailable. The recommendation is to use *static SQL with host variables* or *dynamic SQL without parameter markers* as the most efficient options.

Using PREPARE, DESCRIBE, FETCH and the SQLDA

With static SQL, host variables used in embedded SQL statements are known at application compile time. With dynamic SQL, the embedded SQL statements and consequently the host variables are not known until application run time. Thus, for dynamic SQL applications, you need to deal with the list of host variables that are used in your application. You can use the DESCRIBE statement to obtain host variable information for any SELECT statement that has been prepared (using PREPARE), and store that information into the SQL descriptor area (SQLDA).

Note: Java applications do not use the SQLDA structure, and therefore do not use the PREPARE or DESCRIBE statements. In JDBC applications you can use a PreparedStatement object and the executeQuery() method to generate a ResultSet object, which is the equivalent of a host language cursor. In SQLJ applications you can also declare an SQLJ iterator object with a CursorByPos or CursorByName cursor to return data from FETCH statements.

When the DESCRIBE statement gets executed in your application, the database manager defines your host variables in an SQLDA. Once the host variables are defined in the SQLDA, you can use the FETCH statement to assign values to the host variables, using a cursor.

For complete information on the PREPARE, DESCRIBE, and FETCH statements, and a description of the SQLDA, refer to the *SQL Reference*.

For an example of a simple dynamic SQL program that uses the PREPARE, DESCRIBE, and FETCH statements without using an SQLDA, see "Example:

Dynamic SQL Program” on page 136. For an example of a dynamic SQL program that uses the PREPARE, DESCRIBE, and FETCH statements and an SQLDA to process interactive SQL statements, see “Example: ADHOC Program” on page 159.

Declaring and Using Cursors

Processing a cursor dynamically is nearly identical to processing it using static SQL. When a cursor is declared, it is associated with a query.

In the static SQL case, the query is a SELECT statement in text form, as shown in “Declare Cursor Statement” on page 79.

In the dynamic SQL case, the query is associated with a statement name assigned in a PREPARE statement. Any referenced host variables are represented by parameter markers. Table 7 shows a DECLARE statement associated with a dynamic SELECT statement.

Table 7. Declare Statement Associated with a Dynamic SELECT

Language	Example Source Code
C/C++	<pre>strcpy(prep_string, "SELECT tabname FROM syscat.tables" "WHERE tabschema = ?"); EXEC SQL PREPARE s1 FROM :prep_string; EXEC SQL DECLARE c1 CURSOR FOR s1; EXEC SQL OPEN c1 USING :host_var;</pre>
Java (JDBC)	<pre>PreparedStatement prep_string = ("SELECT tabname FROM syscat.tables WHERE tabschema = ?"); prep_string.setCursor("c1"); prep_string.setString(1, host_var); ResultSet rs = prep_string.executeQuery();</pre>
COBOL	<pre>MOVE "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?" TO PREP-STRING. EXEC SQL PREPARE S1 FROM :PREP-STRING END-EXEC. EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC. EXEC SQL OPEN C1 USING :host-var END-EXEC.</pre>
FORTRAN	<pre>prep_string = 'SELECT tabname FROM syscat.tables WHERE tabschema = ?' EXEC SQL PREPARE s1 FROM :prep_string EXEC SQL DECLARE c1 CURSOR FOR s1 EXEC SQL OPEN c1 USING :host_var</pre>

The main difference between a static and a dynamic cursor is that a static cursor is prepared at precompile time, and a dynamic cursor is prepared at run time. Additionally, host variables referenced in the query are represented by parameter markers, which are replaced by run-time host variables when the cursor is opened.

For more information about how to use cursors, see the following sections:

- “Selecting Multiple Rows Using a Cursor” on page 78
- “Example: Cursor Program” on page 82
- “Using Cursors in REXX” on page 619

Example: Dynamic SQL Program

This sample program shows the processing of a cursor based upon a dynamic SQL statement. It lists all the tables in SYSCAT.TABLES except for the tables with the value STAFF in the name column. The sample is available in the following programming languages:

C	dynamic.sqc
Java	Dynamic.java
COBOL	dynamic.sqb
FORTRAN	dynamic.sqf
REXX	dynamic.cmd

How the Dynamic Program Works

1. **Declare host variables.** This section includes declarations of three host variables:
 - table_name** Used to hold the data returned during the FETCH statement
 - st** Used to hold the dynamic SQL statement in text form
 - parm_var** Supplies a data value to replace the parameter marker in st.
2. **Prepare the statement.** An SQL statement with one parameter marker (indicated by '?') is copied to the host variable. This host variable is passed to the PREPARE statement for validation. The PREPARE statement parses the SQL text and prepares an access section for the package in the same way that the precompiler or binder does, only it happens at run time instead of during preprocessing.
3. **Declare the cursor.** The DECLARE statement associates a cursor with a dynamically prepared SQL statement. If the prepared SQL statement is a SELECT statement, a cursor is necessary to retrieve the rows from the result table.
4. **Open the cursor.** The OPEN statement initializes the cursor declared earlier to point before the first row of the result table. The USING clause specifies a host variable to replace the parameter marker in the prepared SQL statement. The data type and length of the host variable must be compatible with the associated column type and length.
5. **Retrieve the data.** The FETCH statement is used to move the NAME column from the result table into the table_name host variable. The host variable is printed before the program loops back to fetch another row.
6. **Close the cursor.** The CLOSE statement closes the cursor and releases the resources associated with it.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C	check_error is redefined as CHECKERR and is located in the util.c file.
Java	Any SQL error is thrown as an SQLException and handled in the catch block of the application.
COBOL	CHECKERR is an external program named checkerr.cbl.
FORTTRAN	CHECKERR is a subroutine located in the util.f file.
REXX	CHECKERR is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

C Example: DYNAMIC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR)  if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
    char table_name[19];
    char st[80]; 1
    char parm_var[19];
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

#ifdef DB268K
    /* Before making any API calls for 68K environment,
       need to initial the Library Manager */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
#endif

    printf( "Sample C program: DYNAMIC\n" );

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: dynamic [userid passwd]\n\n");
        return 1;
    } /* endif */

    strcpy( st, "SELECT tabname FROM syscat.tables" );
    strcat( st, " WHERE tabname <> ?" );
    EXEC SQL PREPARE s1 FROM :st; 2
    CHECKERR ("PREPARE");

    EXEC SQL DECLARE c1 CURSOR FOR s1; 3
```

```

strcpy( parm_var, "STAFF" );
EXEC SQL OPEN c1 USING :parm_var; 4
CHECKERR ("OPEN");
do {
    EXEC SQL FETCH c1 INTO :table_name; 5
    if (SQLCODE != 0) break;

    printf( "Table = %s\n", table_name );
} while ( 1 );

EXEC SQL CLOSE c1; 6
CHECKERR ("CLOSE");

EXEC SQL COMMIT;
CHECKERR ("COMMIT");

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : DYNAMIC.SQC */

```

Java Example: Dynamic.java

```
import java.sql.*;

class Dynamic
{
    static
    {
        try
        {
            Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {
            System.out.println ("\n Error loading DB2 Driver...\n");
            System.out.println (e);
            System.exit(1);
        }
    }

    public static void main(String argv[])
    {
        try
        {
            System.out.println (" Java Dynamic Sample");
            // Connect to Sample database

            Connection con = null;
            // URL is jdbc:db2:dbname
            String url = "jdbc:db2:sample";

            if (argv.length == 0)
            {
                // connect with default id/password
                con = DriverManager.getConnection(url);
            }
            else if (argv.length == 2)
            {
                String userid = argv[0];
                String passwd = argv[1];

                // connect with user-provided username and password
                con = DriverManager.getConnection(url, userid, passwd);
            }
            else
            {
                throw new Exception("Usage: java Dynamic [username password]");
            }

            // Enable transactions
            con.setAutoCommit(false);

            // Perform dynamic SQL SELECT using JDBC
            try
            {
                PreparedStatement pstmt1 = con.prepareStatement(
                    "SELECT tabname FROM syscat.tables WHERE tabname <> ?"); 2
                // set cursor name for the positioned update statement
                pstmt1.setCursorName("c1"); 3
                pstmt1.setString(1, "STAFF"); 4
                ResultSet rs = pstmt1.executeQuery();

                System.out.print("\n");
                while( rs.next() ) 5
                {
                    String tableName = rs.getString("tabname");
                    System.out.println("Table = " + tableName);
                }
            }
        }
    }
}
```



```

        };

        rs.close();
        pstmt1.close();
    }
    catch( Exception e )
    {
        throw e;
    }
    finally
    {
        // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        con.rollback();
        System.out.println("Rollback done.");
    }
}
catch( Exception e )
{
    System.out.println(e);
}
}
}

```

COBOL Example: DYNAMIC.SQB

Identification Division.
Program-ID. "dynamic".

Data Division.
Working-Storage Section.

copy "sqlenv.cbl".
copy "sql.cbl".
copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 table-name pic x(20).
01 st pic x(80).
01 parm-var pic x(18).
01 userid pic x(8).
01 passwd.
49 passwd-length pic s9(4) comp-5 value 0.
49 passwd-name pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc pic x(80).

1

Procedure Division.
Main Section.

display "Sample COBOL program: DYNAMIC".

display "Enter your user id (default none): "
with no advancing.
accept userid.

if userid = spaces
EXEC SQL CONNECT TO sample END-EXEC
else
display "Enter your password : " with no advancing
accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
inspect passwd-name tallying passwd-length for characters
before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.
move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

move "SELECT TABNAME FROM SYSCAT.TABLES
WHERE TABNAME <> ?" to st.
EXEC SQL PREPARE s1 FROM :st END-EXEC.
move "PREPARE" to errloc.
call "checkerr" using SQLCA errloc.

2

EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

3

```

move "STAFF" to parm-var.
EXEC SQL OPEN c1 USING :parm-var END-EXEC.
move "OPEN" to errloc.
call "checkerr" using SQLCA errloc.

perform Fetch-Loop thru End-Fetch-Loop
until SQLCODE not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
move "CLOSE" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL COMMIT END-EXEC.
move "COMMIT" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.

End-Main.
go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :table-name END-EXEC.
if SQLCODE not equal 0
go to End-Fetch-Loop.
display "TABLE = ", table-name.
End-Fetch-Loop. exit.

End-Prog.
stop run.

```

FORTRAN Example: DYNAMIC.SQF

```
program dynamic
implicit none

include 'sqlenv.f'
EXEC SQL INCLUDE SQLCA

EXEC SQL BEGIN DECLARE SECTION 1
  character*8   userid
  character*18  passwd
  character*20  table_name
  character*80  st
  character*18  parm_var
EXEC SQL END DECLARE SECTION

  character*80  errloc

print *, 'Sample FORTRAN program: DYNAMIC'

print *, 'Enter your user id (default none):'
read 101, userid
101 format (a8)

if( userid(1:1) .eq. ' ' ) then
  EXEC SQL CONNECT TO sample
else
  print *, 'Enter your password :'
  read 101, passwd

  EXEC SQL CONNECT TO sample USER :userid USING :passwd
end if
errloc = 'CONNECT'
call checkerr (sqlca, errloc, *999)

st = 'SELECT tabname FROM syscat.tables WHERE tabname <> ?'
EXEC SQL PREPARE s1 FROM :st 2
errloc = 'PREPARE'
call checkerr (sqlca, errloc, *999)

EXEC SQL DECLARE c1 CURSOR FOR s1 3

parm_var = 'STAFF'
EXEC SQL OPEN c1 USING :parm_var 4
errloc = 'OPEN'
call checkerr (sqlca, errloc, *999)

10 continue
  EXEC SQL FETCH c1 INTO :table_name 5
  if (sqlcode .ne. 0) goto 100

  print *, 'Table = ', table_name
  goto 10

100 EXEC SQL CLOSE c1 6
  errloc = 'CLOSE'
```

```
call checkerr (sqlca, errloc, *999)

EXEC SQL COMMIT
errloc = 'COMMIT'
call checkerr (sqlca, errloc, *999)

EXEC SQL CONNECT RESET
errloc = 'CONNECT RESET'
call checkerr (sqlca, errloc, *999)

999 stop
end
```

REXX Example: DYNAMIC.CMD

```
/* this variable (SYSTEM) must be user defined */
SYSTEM = AIX
if SYSTEM = OS2 then do
  if RxFuncQuery('SQLDBS') <> 0 then
    rcy = RxFuncAdd( 'SQLDBS', 'DB2AR', 'SQLDBS' )

  if RxFuncQuery('SQLEXEC') <> 0 then
    rcy = RxFuncAdd( 'SQLEXEC', 'DB2AR', 'SQLEXEC' )
end

if SYSTEM = AIX then
  rcy = SysAddFuncPkg("db2rex")

/* pull in command line arguments */
parse arg userid passwd .

/* check to see if the proper number of arguments have been passed in */
PARSE ARG dbname userid password .
if ((dbname = "" ) | ,
    (userid <> "" & password = "" ) ,
    ) then do
  SAY "USAGE: dynamic.cmd <dbname> [<userid> <password>]"

  exit -1
end

/* connect to database */
SAY
SAY 'Connect to' dbname
IF password= "" THEN
  CALL SQLEXEC 'CONNECT TO' dbname
ELSE
  CALL SQLEXEC 'CONNECT TO' dbname 'USER' userid 'USING' password

CALL CHECKERR 'Connect to '
SAY "Connected"

say 'Sample REXX program: DYNAMIC'

st = "SELECT tabname FROM syscat.tables WHERE tabname <> ?"
call SQLEXEC 'PREPARE s1 FROM :st' 2
call CHECKERR 'PREPARE'

call SQLEXEC 'DECLARE c1 CURSOR FOR s1' 3
call CHECKERR 'DECLARE'

parm_var = "STAFF"
call SQLEXEC 'OPEN c1 USING :parm_var' 4

do while ( SQLCA.SQLCODE = 0 )
  call SQLEXEC 'FETCH c1 INTO :table_name' 5
  if (SQLCA.SQLCODE = 0) then
    say 'Table = ' table_name
end
```

```

call SQLEXEC 'CLOSE c1'
call CHECKERR 'CLOSE'

call SQLEXEC 'CONNECT RESET'
call CHECKERR 'CONNECT RESET'

CHECKERR:
  arg errloc

  if ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

    /*****\
    * GET ERROR MESSAGE API called *
    \*****/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
end
return 0

```

Declaring the SQLDA

An SQLDA contains a variable number of occurrences of SQLVAR entries, each of which contains a set of fields that describe one column in a row of data as shown in Figure 2. There are two types of SQLVAR entries: base SQLVARs, and secondary SQLVARs. For information about the two types, refer to the *SQL Reference*.

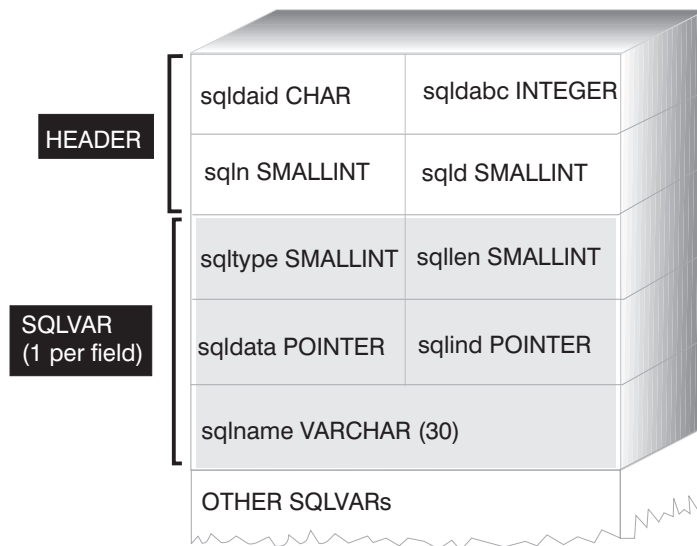


Figure 2. The SQL Descriptor Area (SQLDA)

Since the number of SQLVAR entries required depends on the number of columns in the result table, an application must be able to allocate an appropriate number of SQLVAR elements when needed. Two methods are available as discussed below. For information about the fields of the SQLDA that are mentioned, refer to the *SQL Reference*.

- Provide the largest SQLDA (that is, the one with the greatest number of SQLVAR entries) that is needed. The maximum number of columns that can be returned in a result table is 255. If any of the columns being returned is either a LOB type or a distinct type, the value in SQLN is doubled, and the number of SQLVARs needed to hold the information is doubled to 510. However, as most SELECT statements do not even retrieve 255 columns, most of the allocated space is unused.
- Provide a smaller SQLDA with fewer SQLVAR entries. In this case, if there are more columns in the result than SQLVAR entries allowed for in the SQLDA, then no descriptions are returned. Instead, the database manager returns the number of select list items detected in the SELECT statement. The application allocates an SQLDA with the required number of SQLVAR

entries, and then uses the DESCRIBE statement to acquire the column descriptions. More details on this method are provided in “Preparing the Statement Using the Minimum SQLDA Structure”.

For the above methods, the question arises as to how many initial SQLVAR entries you should allocate. Each SQLVAR element uses up 44 bytes of storage (not counting storage allocated for the SQLDATA and SQLIND fields). If memory is plentiful, the first method of providing an SQLDA of maximum size is easier to implement.

The second method of allocating a smaller SQLDA is only applicable to programming languages such as C and C++ that support the dynamic allocation of memory. For languages such as COBOL and FORTRAN that do not support the dynamic allocation of memory, you have to use the first method.

Preparing the Statement Using the Minimum SQLDA Structure

Suppose an application declares an SQLDA structure named `minsqlda` that contains no SQLVAR entries. The `SQLN` field of the SQLDA describes the number of SQLVAR entries that are allocated. In this case, `SQLN` must be set to 0. Next, to prepare a statement from the character string `dstring` and to enter its description into `minsqlda`, issue the following SQL statement (assuming C syntax, and assuming that `minsqlda` is declared as a pointer to an SQLDA structure):

```
EXEC SQL
  PREPARE STMT INTO :minsqlda FROM :dstring;
```

Suppose that the statement contained in `dstring` was a SELECT statement that returns 20 columns in each row. After the PREPARE statement (or a DESCRIBE statement), the `SQLD` field of the SQLDA contains the number of columns of the result table for the prepared SELECT statement.

The SQLVARs in the SQLDA are set in the following cases:

- `SQLN >= SQLD` and no column is either a LOB or a distinct type
The first `SQLD` SQLVAR entries are set and `SQLDOUBLED` is set to blank.
- `SQLN >= 2*SQLD` and at least one column is a LOB or a distinct type
`2*SQLD` SQLVAR entries are set and `SQLDOUBLED` is set to 2.
- `SQLD <= SQLN < 2*SQLD` and at least one column is a distinct type but there are no LOB columns
The first `SQLD` SQLVAR entries are set and `SQLDOUBLED` is set to blank. If the `SQLWARN` bind option is YES, a warning `SQLCODE +237` (`SQLSTATE 01594`) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no column is either a LOB or distinct type
No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.
Allocate SQLD SQLVARs for a successful DESCRIBE.
- SQLN < SQLD and at least one column is a distinct type but there are no LOB columns
No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.
Allocate 2*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct types.
- SQLN < 2*SQLD and at least one column is a LOB
No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).
Allocate 2*SQLD SQLVARs for a successful DESCRIBE.

The SQLWARN option of the BIND command is used to control whether the DESCRIBE (or PREPARE...INTO) will return the following warnings:

- SQLCODE +236 (SQLSTATE 01005)
- SQLCODE +237 (SQLSTATE 01594)
- SQLCODE +239 (SQLSTATE 01005).

It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 (SQLSTATE 01005) is always returned when there are LOB columns in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB column in the result set.

Allocating an SQLDA with Sufficient SQLVAR Entries

After the number of columns in the result table is determined, storage can be allocated for a second, full-size SQLDA. For example, if the result table contains 20 columns (none of which are LOB columns), a second SQLDA structure, `fulsqlda`, must be allocated with at least 20 SQLVAR elements (or 40 elements if the result table contains any LOBs or distinct types). For the rest of this example, assume that no LOBs or distinct types are in the result table.

The storage requirements for SQLDA structures consist of the following:

- A fixed-length header, 16 bytes in length, containing fields such as SQLN and SQLD
- A varying-length array of SQLVAR entries, of which each element is 44 bytes in length.

The number of SQLVAR entries needed for `fulsqlda` was specified in the SQLD field of `minsqlda`. This value was 20. Therefore, the storage allocation required for `fulsqlda` used in this example is:

$$16 + (20 * 44)$$

This value represents the size of the header plus 20 times the size of each SQLVAR entry, giving a total of 896 bytes.

You can use the `SQLDASIZE` macro to avoid doing your own calculations and to avoid any version-specific dependencies.

Describing the SELECT Statement

Having allocated sufficient space for `fulsqlda`, an application must take the following steps:

1. Store the value 20 in the SQLN field of `fulsqlda`.
2. Obtain information about the SELECT statement using the second SQLDA structure, `fulsqlda`. Two methods are available:
 - Use another PREPARE statement specifying `fulsqlda` instead of `minsqlda`.
 - Use the DESCRIBE statement specifying `fulsqlda`.

Using the DESCRIBE statement is preferred because the costs of preparing the statement a second time are avoided. The DESCRIBE statement simply reuses information previously obtained during the prepare operation to fill in the new SQLDA structure. The following statement can be issued:

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

After this statement is executed, each SQLVAR element contains a description of one column of the result table.

Acquiring Storage to Hold a Row

Before fetching any rows of the result table using an SQLDA structure, an application must do the following:

1. Analyze each SQLVAR description to determine how much space is required for the value of that column.

Note that for Large Object (LOB) values, when the SELECT is described, the data type given in the SQLVAR is `SQL_TYP_xLOB`. This data type corresponds to a plain LOB host variable, that is, the whole LOB will be

stored in memory at one time. This will work for small LOBs (up to a few MB), but you cannot use this data type for large LOBs (say 1 GB). It will be necessary for your application to change its column definition in the SQLVAR to be either SQL_TYP_xLOB_LOCATOR or SQL_TYPE_xLOB_FILE. (Note that changing the SQLTYPE field of the SQLVAR also necessitates changing the SQLLEN field.) After changing the column definition in the SQLVAR, your application can then allocate the correct amount of storage for the new type. For more information on LOBs, see “Chapter 7. Using the Object-Relational Capabilities” on page 221.

2. Allocate storage for the value of that column.
3. Store the address of the allocated storage in the SQLDATA field of the SQLDA structure.

These steps are accomplished by analyzing the description of each column and replacing the content of each SQLDATA field with the address of a storage area large enough to hold any values from that column. The length attribute is determined from the SQLLEN field of each SQLVAR entry for data items that are not of a LOB type. For items with a type of BLOB, CLOB, or DBCLOB, the length attribute is determined from the SQLLONGLEN field of the secondary SQLVAR entry.

In addition, if the specified column allows nulls, then the application must replace the content of the SQLIND field with the address of an indicator variable for the column.

Processing the Cursor

After the SQLDA structure is properly allocated, the cursor associated with the SELECT statement can be opened and rows can be fetched by specifying the USING DESCRIPTOR clause of the FETCH statement.

When finished, the cursor should be closed and any dynamically allocated memory should be released.

Allocating an SQLDA Structure

To create an SQLDA structure with C, either embed the INCLUDE SQLDA statement in the host language or include the SQLDA include file to get the structure definition. Then, because the size of an SQLDA is not fixed, the application must declare a pointer to an SQLDA structure and allocate storage for it. The actual size of the SQLDA structure depends on the number of distinct data items being passed using the SQLDA. (For an example of how to code an application to process the SQLDA, see “Example: ADHOC Program” on page 159.)

In the C/C++ programming language, a macro is provided to facilitate SQLDA allocation. This macro has the following format:

```
#define SQLDASIZE(n) (sizeof(struct sqllda) + (n-1) * sizeof(struct sqlvar))
```

The effect of this macro is to calculate the required storage for an SQLDA with *n* SQLVAR elements.

To create an SQLDA structure with COBOL, you can either embed an INCLUDE SQLDA statement or use the COPY statement. Use the COPY statement when you want to control the maximum number of SQLVARs and hence the amount of storage that the SQLDA uses. For example, to change the default number of SQLVARs from 1489 to 1, use the following COPY statement:

```
COPY "sqllda.cbl"  
  replacing --1489--  
  by --1--.
```

The FORTRAN language does not directly support self-defining data structures or dynamic allocation. No SQLDA include file is provided for FORTRAN, because it is not possible to support the SQLDA as a data structure in FORTRAN. The precompiler will ignore the INCLUDE SQLDA statement in a FORTRAN program.

However, you can create something similar to a static SQLDA structure in a FORTRAN program, and use this structure wherever an SQLDA can be used. The file `sqldact.f` contains constants that help in declaring an SQLDA structure in FORTRAN.

Execute calls to SQLGADDR to assign pointer values to the SQLDA elements that require them.

The following table shows the declaration and use of an SQLDA structure with one SQLVAR element.

Language	Example Source Code
C/C++	<pre> #include <sqlda.h> struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1)); /* DECLARE LOCAL VARIABLES FOR HOLDING ACTUAL DATA */ double sal; short salind; /* INITIALIZE ONE ELEMENT OF SQLDA */ memcpy(outda->sqldaid,"SQLDA ",sizeof(outda->sqldaid)); outda->sqln = outda->sqld = 1; outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT; outda->sqlvar[0].sqlllen = sizeof(double); outda->sqlvar[0].sqldata = (unsigned char *)&sal; outda->sqlvar[0].sqlind = (short *)&salind; </pre>
COBOL	<pre> WORKING-STORAGE SECTION. 77 SALARY PIC S99999V99 COMP-3. 77 SAL-IND PIC S9(4) COMP-5. EXEC SQL INCLUDE SQLDA END-EXEC * Or code a useful way to save unused SQLVAR entries. * COPY "sqlda.cbl" REPLACING --1489-- BY --1--. 01 decimal-sqlllen pic s9(4) comp-5. 01 decimal-parts redefines decimal-sqlllen. 05 precision pic x. 05 scale pic x. * Initialize one element of output SQLDA MOVE 1 TO SQLN MOVE 1 TO SQLD MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1) * Length = 7 digits precision and 2 digits scale MOVE x"07" TO PRECISION. MOVE x"02" TO SCALE. MOVE DECIMAL-SQLLEN TO O-SQLLEN(1). SET SQLDATA(1) TO ADDRESS OF SALARY SET SQLIND(1) TO ADDRESS OF SAL-IND </pre>

Language	Example Source Code
FORTTRAN	<pre> include 'sqldact.f' integer*2 sqlvar1 parameter (sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz) C Declare an Output SQLDA -- 1 Variable character out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz) character*8 out_sqldaaid ! Header integer*4 out_sqldabc integer*2 out_sqln integer*2 out_sqld integer*2 out_sqltype1 ! First Variable integer*2 out_sqllen1 integer*4 out_sqldata1 integer*4 out_sqlind1 integer*2 out_sqlnamel1 character*30 out_sqlnamec1 equivalence(out_sqlda(sqlda_sqldaaid_ofs), out_sqldaaid) equivalence(out_sqlda(sqlda_sqldabc_ofs), out_sqldabc) equivalence(out_sqlda(sqlda_sqln_ofs), out_sqln) equivalence(out_sqlda(sqlda_sqld_ofs), out_sqld) equivalence(out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1) equivalence(out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1) equivalence(out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1) equivalence(out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1) equivalence(out_sqlda(sqlvar1+sqlvar_name_length_ofs), + out_sqlnamel1) equivalence(out_sqlda(sqlvar1+sqlvar_name_data_ofs), + out_sqlnamec1) C Declare Local Variables for Holding Returned Data. real*8 salary integer*2 sal_ind C Initialize the Output SQLDA (Header) out_sqldaaid = 'OUT_SQLDA' out_sqldabc = sqlda_header_sz + 1*sqlvar_struct_sz out_sqln = 1 out_sqld = 1 C Initialize VAR1 out_sqltype1 = SQL_TYP_NFLOAT out_sqllen1 = 8 rc = sqlgaddr(%ref(salary), %ref(out_sqldata1)) rc = sqlgaddr(%ref(sal_ind), %ref(out_sqlind1)) </pre>

In languages not supporting dynamic memory allocation, an SQLDA with the desired number of SQLVAR elements must be explicitly declared in the host language. Be sure to declare enough SQLVAR elements as determined by the needs of the application.

Passing Data Using an SQLDA Structure

Greater flexibility is available when passing data using an SQLDA than is available using lists of host variables. For example, an SQLDA can be used to transfer data that has no native host language equivalent, such as DECIMAL data in the C language. The sample program called ADHOC is an example using this technique. (See “Example: ADHOC Program” on page 159.) See Table 8 for a convenient cross-reference listing showing how the numeric values and symbolic names are related.

Table 8. DB2 V2 SQLDA SQL Types. Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name ¹
DATE	384/385	SQL_TYP_DATE / SQL_TYP_NDATE
TIME	388/389	SQL_TYP_TIME / SQL_TYP_NTIME
TIMESTAMP	392/393	SQL_TYP_STAMP / SQL_TYP_NSTAMP
n/a ²	400/401	SQL_TYP_CGSTR / SQL_TYP_NCGSTR
BLOB	404/405	SQL_TYP_BLOB / SQL_TYP_NBLOB
CLOB	408/409	SQL_TYP_CLOB / SQL_TYP_NCLOB
DBCLOB	412/413	SQL_TYP_DBCLOB / SQL_TYP_NDBCLOB
VARCHAR	448/449	SQL_TYP_VARCHAR / SQL_TYP_NVARCHAR
CHAR	452/453	SQL_TYP_CHAR / SQL_TYP_NCHAR
LONG VARCHAR	456/457	SQL_TYP_LONG / SQL_TYP_NLONG
n/a ³	460/461	SQL_TYP_CSTR / SQL_TYP_NCSTR
VARGRAPHIC	464/465	SQL_TYP_VARGRAPH / SQL_TYP_NVARGRAPH
GRAPHIC	468/469	SQL_TYP_GRAPHIC / SQL_TYP_NGRAPHIC
LONG VARGRAPHIC	472/473	SQL_TYP_LONGGRAPH / SQL_TYP_NLONGGRAPH
FLOAT	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
REAL ⁴	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
DECIMAL ⁵	484/485	SQL_TYP_DECIMAL / SQL_TYP_DECIMAL
INTEGER	496/497	SQL_TYP_INTEGER / SQL_TYP_NINTEGER
SMALLINT	500/501	SQL_TYP_SMALL / SQL_TYP_NSMALL
n/a	804/805	SQL_TYP_BLOB_FILE / SQL_TYP_NBLOB_FILE
n/a	808/809	SQL_TYP_CLOB_FILE / SQL_TYP_NCLOB_FILE
n/a	812/813	SQL_TYP_DBCLOB_FILE / SQL_TYP_NDBCLOB_FILE
n/a	960/961	SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR
n/a	964/965	SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR

Table 8. DB2 V2 SQLDA SQL Types (continued). Numeric Values and Corresponding Symbolic Names

SQL Column Type	SQLTYPE numeric value	SQLTYPE symbolic name ¹
n/a	968/969	SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR

Note: These defined types can be found in the `sql.h` include file located in the `include` sub-directory of the `sqllib` directory. (For example, `sqllib/include/sql.h` for the C programming language.)

1. For the COBOL programming language, the SQLTYPE name does not use underscore (`_`) but uses a hyphen (`-`) instead.
2. This is a null-terminated graphic string.
3. This is a null-terminated character string.
4. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
5. Precision is in the first byte. Scale is in the second byte.

Processing Interactive SQL Statements

An application using dynamic SQL can be written to process arbitrary SQL statements. For example, if an application accepts SQL statements from a user, the application must be able to execute the statements without any prior knowledge of the statements.

By using the PREPARE and DESCRIBE statements with an SQLDA structure, an application can determine the type of SQL statement being executed, and act accordingly.

For an example of a program that processes interactive SQL statements, see “Example: ADHOC Program” on page 159.

Determining Statement Type

When an SQL statement is prepared, information concerning the type of statement can be determined by examining the SQLDA structure. This information is placed in the SQLDA structure either at statement preparation time with the INTO clause, or by issuing a DESCRIBE statement against a previously prepared statement.

In either case, the database manager places a value in the SQLD field of the SQLDA structure, indicating the number of columns in the result table generated by the SQL statement. If the SQLD field contains a zero (0), the statement is *not* a SELECT statement. Since the statement is already prepared, it can immediately be executed using the EXECUTE statement.

If the statement contains parameter markers, the USING clause must be specified as described in the *SQL Reference*. The USING clause can specify either a list of host variables or an SQLDA structure.

If the SQLD field is greater than zero, the statement is a SELECT statement and must be processed as described in the following sections.

Varying-List SELECT Statement

A *varying-list* SELECT statement is one in which the number and types of columns that are to be returned are not known at precompilation time. In this case, the application does not know in advance the exact host variables that need to be declared to hold a row of the result table.

To process a varying-list SELECT statement, an application can do the following:

1. **Declare an SQLDA.** An SQLDA structure must be used to process varying-list SELECT statements.
2. **PREPARE the statement using the INTO clause.** The application then determines whether the SQLDA structure declared has enough SQLVAR elements. If it does not, the application allocates another SQLDA structure with the required number of SQLVAR elements, and issues an additional DESCRIBE statement using the new SQLDA.
3. **Allocate the SQLVAR elements.** Allocate storage for the host variables and indicators needed for each SQLVAR. This step involves placing the allocated addresses for the data and indicator variables in each SQLVAR element.
4. **Process the SELECT statement.** A cursor is associated with the prepared statement, opened, and rows are fetched using the properly allocated SQLDA structure.

These steps are described in detail in the following sections:

- “Declaring the SQLDA” on page 148
- “Preparing the Statement Using the Minimum SQLDA Structure” on page 149
- “Allocating an SQLDA with Sufficient SQLVAR Entries” on page 150
- “Describing the SELECT Statement” on page 151
- “Acquiring Storage to Hold a Row” on page 151
- “Processing the Cursor” on page 152.

Saving SQL Requests from End Users

If your application allows users to save arbitrary SQL statements, you can save them in a table with a column having a data type of VARCHAR, LONG VARCHAR, CLOB, VARGRAPHIC, LONG VARGRAPHIC or DBCLOB. Note

that the VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB data types are only available in Double Byte Character Support (DBCS) and Extended UNIX Code (EUC) environments.

You must save the source SQL statements, not the prepared versions. This means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your application prepares an SQL statement from a character string and executes this statement dynamically.

Example: ADHOC Program

This sample program shows how the SQLDA is used to process interactive SQL statements.

Note: The example `adhoc.sqc` exists for C only.

How the ADHOC Program Works

1. **Define an SQLDA structure.** The `INCLUDE SQLDA` statement defines and declares an SQLDA structure, which is used to pass data from the database manager to the program and back.
2. **Define an SQLCA structure.** The `INCLUDE SQLCA` statement defines an SQLCA structure, and defines `SQLCODE` as an element within the structure. The `SQLCODE` field of the SQLCA structure is updated with diagnostic information by the database manager after execution of SQL statements.
3. **Declare host variables.** The `BEGIN DECLARE SECTION` and `END DECLARE SECTION` statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement.
4. **Connect to database.** The program connects to the database specified by the user, and requests shared access to it. (It is assumed that a `START DATABASE MANAGER API` call or `db2start` command has been issued.) Other programs that attempt to connect to the same database in share mode are also granted access.
5. **Check completion.** The SQLCA structure is checked for successful completion of the `CONNECT TO` statement. An `SQLCODE` value of 0 indicates that the connection was successful.
6. **Interactive prompt.** SQL statements are entered in through the prompt and then are sent to the `process_statement` function for further processing.
7. **End the transaction - COMMIT.** The unit of work is ended with a `COMMIT` if so chosen by the user. All changes requested by the SQL statements entered since this last `COMMIT` are saved in the database.

8. **End the transaction - ROLLBACK.** The unit of work is ended with a ROLLBACK if so chosen by the user. All changes requested by the SQL statements entered since the last COMMIT or the start of the program, are undone.
9. **Disconnect from the database.** The program disconnects from the database by executing the CONNECT RESET statement. Upon return, the SQLCA is checked for successful completion.
10. **Copy SQL statement text to host variable.** The statement text is copied into the data area specified by the host variable st.
11. **Prepare the SQLDA for processing.** An initial SQLDA structure is declared and memory is allocated through the init_da procedure to determine what type of output the SQL statement could generate. The SQLDA returned from this PREPARE statement reports the number of columns that will be returned from the SQL statement.
12. **SQLDA reports output columns exist.** The SQL statement is a SELECT statement. The SQLDA is initialized through the init_da procedure to allocate memory space for the prepared SQL statement to reside in.
13. **SQLDA reports no output columns.** There are no columns to be returned. The SQL statement is executed *dynamically* using the EXECUTE statement.
14. **Preparing memory space for the SQLDA.** Memory is allocated to reflect the column structures in the SQLDA. The required amount of memory is selected by the SQLTYPE and the SQLLEN of the column structure in the SQLDA.
15. **Declare and open a cursor.** The DECLARE statement associates the cursor pcurs with the dynamically prepared SQL statement in sqlStatement and the cursor is opened.
16. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the SQLDA.
17. **Display the column titles.** The first row that is fetched is the column title information.
18. **Display the row information.** The rows of information collected from each consecutive FETCH is displayed.
19. **Close the cursor.** The CLOSE statement is closes the cursor, and releases the resources associated with it.

The CHECKERR macro/function is an error checking utility which is external to this program. In this program, check_error is redefined as CHECKERR and is located in the util.c file. See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

Note that this example uses a number of additional procedures that are provided as utilities in the file util.c. These include:

get_info

Returns the input from a passed parameter but only if the input conforms to a specified sizing restriction, thus it is used to verify input retrieval.

init_da

Allocates memory for a prepared SQL statement. An internally described function called SQLDASIZE is used to calculate the proper amount of memory.

alloc_host_vars

Allocates memory for data from an SQLDA pointer.

free_da

Frees up the memory that has been allocated to use an SQLDA data structure.

print_var

Prints out the SQLDA SQLVAR variables. This procedure first determines data type then calls the appropriate subroutines that are required to print out the data.

display_da

Displays the output of a pointer that has been passed through. All pertinent information on the structure of the output data is available from this pointer, as examined in the procedure print_var.

C Example: ADHOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlcodes.h>
#include <sqlda.h> 1
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA ; 2

/* 'check_error' is a function found in the util.c program */
#define CHECKERR(CE_STRING) if (check_error(CE_STRING,&sqlca)!= 0) return -1 ;
#define SQLSTATE sqlca.sqlstate

int process_statement( char * ) ;

int main( int argc, char *argv[] ) {

    int rc ;
    char sqlInput[256] ;
    char st[1024] ;

    EXEC SQL BEGIN DECLARE SECTION ; 3
        char userid[9] ;
        char passwd[19] ;
    EXEC SQL END DECLARE SECTION ;

#ifdef DB268K
/*
Before making any API calls for 68K environment,
need to initial the Library Manager
*/
InitLibraryManager(0,kCurrentZone,kNormalMemory) ;
atexit(CleanupLibraryManager) ;
#endif

    printf( "Sample C program : ADHOC interactive SQL\n" ) ;

    /* Initialize the connection to a database. */
    if ( argc == 1 ) {
        EXEC SQL CONNECT TO sample ;
        CHECKERR( "CONNECT TO SAMPLE" ) ;
    }
    else if ( argc == 3 ) {
        strcpy( userid, argv[1] ) ;
        strcpy( passwd, argv[2] ) ;
        EXEC SQL CONNECT TO sample USER :userid USING :passwd ; 4
        CHECKERR( "CONNECT TO SAMPLE" ) ; 5
    }
}
```

```

    }
    else {
        printf( "\nUSAGE: adhoc [userid passwd]\n\n" );
        return( 1 );
    } /* endif */

    printf( "Connected to database SAMPLE\n" );

    /* Enter the continuous command line loop. */
    *sqlInput = '\0' ;
    while ( ( *sqlInput != 'q' ) && ( *sqlInput != 'Q' ) ) { 6

        printf( "Enter an SQL statement or 'quit' to Quit :\n" );
        gets( sqlInput );

        if ( ( *sqlInput == 'q' ) || ( *sqlInput == 'Q' ) ) break ;

        if ( *sqlInput == '\0' ) { /* Don't process the statement */
            printf( "No characters entered.\n" );
            continue ;
        }

        strcpy( st, sqlInput );
        while ( sqlInput[strlen( sqlInput ) - 1] == '\\' ) {
            st[strlen( st ) - 1] = '\0' ;
            gets( sqlInput );
            strcat( st, sqlInput );
        }

        /* Process the statement. */
        rc = process_statement( st );

    }

    printf( "Enter 'c' to COMMIT or Any Other key to
            ROLLBACK the transaction :\n" );
    gets( sqlInput );
    if ( ( *sqlInput == 'c' ) || ( *sqlInput == 'C' ) ) {
        printf( "COMMITTING the transactions.\n" );
        EXEC SQL COMMIT ; 7
        CHECKERR( "COMMIT" );
    }
    else { /* assume that the transaction is to be rolled back */
        printf( "ROLLING BACK the transactions.\n" );
        EXEC SQL ROLLBACK ; 8
        CHECKERR( "ROLLBACK" );
    }

    EXEC SQL CONNECT RESET ; 9
    CHECKERR( "CONNECT RESET" );

    return( 0 );
}

```

```

/*****
 * FUNCTION : process_statement
 * This function processes the inputted statement and then prepares the
 * procedural SQL implementation to take place.
 *****/
int process_statement ( char * sqlInput ) {

    int counter = 0 ;
    struct sqlda * sqldaPointer ;
    short sqlda_d ;

    EXEC SQL BEGIN DECLARE SECTION ; 3
        char st[1024] ;
    EXEC SQL END DECLARE SECTION ;

    strcpy( st, sqlInput ) ; 10
    /* allocate an initial SQLDA temp pointer to obtain information
       about the inputted "st" */

    init_da( &sqldaPointer, 1 ) ; 11

    EXEC SQL PREPARE statement1 from :st ;
    CHECKERR( "PREPARE" ) ;

    EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer ;

    /* Expecting a return code of 0 or SQL_RC_W236, SQL_RC_W237,
       SQL_RC_W238, SQL_RC_W239 for cases where this statement
       is a SELECT statement. */
    if ( SQLCODE != 0          &&
        SQLCODE != SQL_RC_W236 &&
        SQLCODE != SQL_RC_W237 &&
        SQLCODE != SQL_RC_W238 &&
        SQLCODE != SQL_RC_W239
    ) {
        /* An unexpected warning/error has occurred. Check the SQLCA. */
        CHECKERR( "DESCRIBE" ) ;
    } /* end if */

    sqlda_d = sqldaPointer->sql_d ;
    free( sqldaPointer ) ;

    if ( sqlda_d > 0 ) { 12

        /* this is a SELECT statement, a number of columns
           are present in the SQLDA */

        if ( SQLCODE == SQL_RC_W236 || SQLCODE == 0 )
            /* this out only needs a SINGLE SQLDA */
            init_da( &sqldaPointer, sqlda_d ) ;

        if ( SQLCODE == SQL_RC_W237 || SQLCODE == SQL_RC_W238
            || SQLCODE == SQL_RC_W239 )
            /* this output contains columns that need a DOUBLED SQLDA */
            init_da( &sqldaPointer, sqlda_d * 2 ) ;
    }
}

```



```

/* need to reassign the SQLDA with the correct
   number of columns to the SQL statement */
EXEC SQL DESCRIBE statement1 INTO :sqldaPointer ;
CHECKERR( "DESCRIBE" ) ;

/* allocating the proper amount of memory space
   needed for the variables */
alloc_host_vars( sqldaPointer ) ; 14

/* Don't need to check the SQLCODE for declaration of cursors */
EXEC SQL DECLARE pcurs CURSOR FOR statement1 ; 15

EXEC SQL OPEN pcurs ; 15
CHECKERR( "OPEN" ) ;

EXEC SQL FETCH pcurs USING DESCRIPTOR :sqldaPointer; 16
CHECKERR( "FETCH" ) ;

/* if the FETCH is successful, obtain data from SQLDA */
/* display the column titles */
display_col_titles( sqldaPointer ) ; 17

/* display the rows that are fetched */
while ( SQLCODE == 0 ) {
    counter++ ;
    display_da( sqldaPointer ) ; 18
    EXEC SQL FETCH pcurs USING DESCRIPTOR :sqldaPointer ;
} /* endwhile */

EXEC SQL CLOSE pcurs ; 19
CHECKERR( "CLOSE CURSOR" ) ;
printf( "\n %d record(s) selected\n\n", counter ) ;

/* Free the memory allocated to this SQLDA. */
free_da( sqldaPointer ) ;

} else { /* this is not a SELECT statement,
           execute SQL statement */ 13
    EXEC SQL EXECUTE statement1 ;
    CHECKERR( "Executing the SQL statement" ) ;
} /* end if */

return( 0 ) ;

} /* end of program : ADHOC.SQC */

```

Variable Input to Dynamic SQL

This section shows you how to use parameter markers in your dynamic SQL applications to represent host variable information. It includes:

- Using Parameter Markers
- Example: VARINP Program

Using Parameter Markers

A dynamic SQL statement cannot contain host variables, because host variable information (data type and length) is available only during application precompilation. At execution time, the host variable information is not present. Therefore, a new method is needed to represent application variables. Host variables are represented by a question mark (?) which is called a *parameter marker*. Parameter markers indicate the places in which a host variable is to be substituted inside of an SQL statement. The parameter marker takes on an assumed data type and length that is dependent on the context of its use inside the SQL statement.

If the data type of a parameter marker is not obvious from the context of the statement in which it is used, the type can be specified using a CAST. Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like a host variable of the given type. For example, the statement `SELECT ? FROM SYSCAT.TABLES` is invalid because DB2 does not know the type of the result column. However, the statement `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES`, is valid because the cast *promises* that the parameter marker represents an INTEGER, so DB2 knows the type of the result column.

A character string containing a parameter marker might look like the following:

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

When this statement is executed, a host variable or SQLDA structure is specified by the USING clause of the EXECUTE statement. The contents of the host variable are used when the statement executes.

If the SQL statement contains more than one parameter marker, then the USING clause of the EXECUTE statement must either specify a list of host variables (one for each parameter marker), or it must identify an SQLDA that has an SQLVAR entry for each parameter marker. (Note that for LOBs, there are two SQLVARs per parameter marker.) The host variable list or SQLVAR entries are matched according to the order of the parameter markers in the statement, and they must have compatible data types.

Note that using a parameter marker with dynamic SQL is like using host variables with static SQL. In either case, the optimizer does not use distribution statistics, and possibly may not choose the best access plan.

The rules that apply to parameter markers are listed under the PREPARE statement in the *SQL Reference*.

Example: VARINP Program

This is an example of an UPDATE that uses a parameter marker in the search and update conditions. The sample is available in the following programming languages:

C	varinp.sqc
Java	Varinp.java
COBOL	varinp.sqb
FORTRAN	varinp.sqf

How the VARINP Program Works

1. **Prepare the SELECT SQL statement** The PREPARE statement is called to dynamically prepare an SQL statement. In this SQL statement, parameter markers are denoted by the ?. The job field of staff is defined to be updatable, even though it is not specified in the result table.
2. **Declare the cursor.** The DECLARE CURSOR statement associates the cursor c1 to the query that was prepared in **1**.
3. **Open the cursor.** The cursor c1 is opened, causing the database manager to perform the query and build a result table. The cursor is positioned *before* the first row.
4. **Prepare the UPDATE SQL statement** The PREPARE statement is called to dynamically prepare an SQL statement. The parameter marker in this statement is set to be Clerk but can be changed dynamically to anything, as long as it conforms to the column data type it is being updated into.
5. **Retrieve a row.** The FETCH statement positions the cursor at the next row and moves the contents of the row into the host variables. This row becomes the *CURRENT* row.
6. **Update the current row.** The current row and specified column, job, is updated with the content of the passed parameter parm_var.
7. **Close the cursor.** The CLOSE statement is issued, releasing the resources associated with the cursor. The cursor can be opened again, however.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C	check_error is redefined as CHECKERR and is located in the util.c file.
Java	Any SQL error is thrown as an SQLException and handled in the catch block of the application.
COBOL	CHECKERR is an external program named checkerr.cbl
FORTTRAN	CHECKERR is a subroutine located in the util.f file.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

C Example: VARINP.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR)  if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
        char  pname[10];
        short dept;
        char userid[9];
        char passwd[19];
        char st[255];
        char parm_var[6];
    EXEC SQL END DECLARE SECTION;

#ifdef DB268K
    /* Before making any API calls for 68K environment,
       need to initial the Library Manager */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
#endif

    printf( "Sample C program: VARINP \n" );

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: varinp [userid passwd]\n\n");
        return 1;
    } /* endif */

    strcpy (st, "SELECT name, dept FROM staff WHERE job = ? FOR UPDATE OF job");
    EXEC SQL PREPARE s1 FROM :st; 1
    CHECKERR ("PREPARE");

    EXEC SQL DECLARE c1 CURSOR FOR s1; 2
```

```

strcpy (parm_var, "Mgr");
EXEC SQL OPEN c1 USING :parm_var; 3
CHECKERR ("OPEN");

strcpy (parm_var, "Clerk");
strcpy (st, "UPDATE staff SET job = ? WHERE CURRENT OF c1");
EXEC SQL PREPARE s2 from :st; 4

do {
    EXEC SQL FETCH c1 INTO :pname, :dept; 5
    if (SQLCODE != 0) break;

    printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
    EXEC SQL EXECUTE s2 USING :parm_var; 6
    CHECKERR ("EXECUTE");
} while ( 1 );

EXEC SQL CLOSE c1; 7
CHECKERR ("CLOSE CURSOR");

EXEC SQL ROLLBACK;
CHECKERR ("ROLLBACK");
printf( "\nOn second thought -- changes rolled back.\n" );

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : VARINP.SQC */

```

Java Example: Varinp.java

```
import java.sql.*;

class Varinp
{
    static
    {
        try
        {
            Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {
            System.out.println ("\n Error loading DB2 Driver...\n");
        }
        System.out.println (e);
        System.exit(1);
    }
}

public static void main(String argv[])
{
    try
    {
        System.out.println (" Java Varinp Sample");
        // Connect to Sample database

        Connection con = null;
        // URL is jdbc:db2:dbname
        String url = "jdbc:db2:sample";

        if (argv.length == 0)
        {
            // connect with default id/password
            con = DriverManager.getConnection(url);
        }
        else if (argv.length == 2)
        {
            String userid = argv[0];
            String passwd = argv[1];

            // connect with user-provided username and password
            con = DriverManager.getConnection(url, userid, passwd);
        }
        else
        {
            throw new Exception("\nUsage: java Varinp [username password]\n");
        }

        // Enable transactions
        con.setAutoCommit(false);

        // Perform dynamic SQL using JDBC
        try
        {
            PreparedStatement pstmt1 = con.prepareStatement(
                "SELECT name, dept FROM staff WHERE job = ? FOR UPDATE OF job"); 1
            // set cursor name for the positioned update statement
            pstmt1.setCursorName("c1"); 2
            pstmt1.setString(1, "Mgr");
            ResultSet rs = pstmt1.executeQuery(); 3

            PreparedStatement pstmt2 = con.prepareStatement(
                "UPDATE staff SET job = ? WHERE CURRENT OF c1"); 4
            pstmt2.setString(1, "Clerk");
        }
        catch (Exception e)
        {
            System.out.println (e);
            System.exit(1);
        }
    }
}
```

```

        System.out.print("\n");
        while( rs.next() ) 5
        {
            String name = rs.getString("name");
            short dept = rs.getShort("dept");
            System.out.println(name + " in dept. " +
            dept + " will be demoted to Clerk");

            pstmt2.executeUpdate(); 6
        };

        rs.close();
        pstmt1.close(); 7
        pstmt2.close();
    }
    catch( Exception e )
    {
        throw e;
    }
    finally
    {
        // Rollback the transaction
        System.out.println("\nRollback the transaction...");
        con.rollback();
        System.out.println("Rollback done.");
    }
}
catch( Exception e )
{
    System.out.println(e);
}
}
}

```


COBOL Example: VARINP.SQB

Identification Division.
Program-ID. "varinp".

Data Division.
Working-Storage Section.

copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 pname pic x(10).
01 dept pic s9(4) comp-5.
01 st pic x(127).
01 parm-var pic x(5).
01 userid pic x(8).
01 passwd.
49 passwd-length pic s9(4) comp-5 value 0.
49 passwd-name pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc pic x(80).

Procedure Division.
Main Section.

display "Sample COBOL program: VARINP".

* Get database connection information.
display "Enter your user id (default none): "
with no advancing.
accept userid.

if userid = spaces
EXEC SQL CONNECT TO sample END-EXEC
else
display "Enter your password : " with no advancing
accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR format
* with the length of the input string.
inspect passwd-name tallying passwd-length for characters
before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.
move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

move "SELECT name, dept FROM staff
- " WHERE job = ? FOR UPDATE OF job" to st.
EXEC SQL PREPARE s1 FROM :st END-EXEC.
move "PREPARE" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

1

2

```

move "Mgr" to parm-var.

EXEC SQL OPEN c1 USING :parm-var END-EXEC
move "OPEN" to errloc.
call "checkerr" using SQLCA errloc.

move "Clerk" to parm-var.
move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.

EXEC SQL PREPARE s2 from :st END-EXEC.
move "PREPARE S2" to errloc.
call "checkerr" using SQLCA errloc.

* call the FETCH and UPDATE loop.
  perform Fetch-Loop thru End-Fetch-Loop
    until SQLCODE not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
move "CLOSE" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL ROLLBACK END-EXEC.
move "ROLLBACK" to errloc.
call "checkerr" using SQLCA errloc.
DISPLAY "On second thought -- changes rolled back.".

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Main.
go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :pname, :dept END-EXEC.
if SQLCODE not equal 0
  go to End-Fetch-Loop.
display pname, " in dept. ", dept,
  " will be demoted to Clerk".

EXEC SQL EXECUTE s2 USING :parm-var END-EXEC.
move "EXECUTE" to errloc.
call "checkerr" using SQLCA errloc.

End-Fetch-Loop. exit.

End-Prog.
stop run.

```

FORTRAN Example: VARINP.SQF

```
program varinp
implicit none

include 'sqlenv.f'
EXEC SQL INCLUDE SQLCA

EXEC SQL BEGIN DECLARE SECTION 1
  character*8   userid
  character*18  passwd
  character*9   pname
  integer*4     dept
  character*127 st
  character*5   parm_var
EXEC SQL END DECLARE SECTION

  character*80  errloc

print *, 'Sample FORTRAN program: VARINP'

print *, 'Enter your user id (default none):'
read 109, userid
109 format (a8)

if( userid(1:1) .eq. ' ' ) then
  EXEC SQL CONNECT TO sample
else
  print *, 'Enter your password :'
  read 109, passwd

  EXEC SQL CONNECT TO sample USER :userid USING :passwd
end if
errloc = 'CONNECT'
call checkerr (sqlca, errloc, *999)

st='SELECT name, dept FROM staff WHERE job=? FOR UPDATE of job'
EXEC SQL PREPARE s1 FROM :st 1
errloc = 'PREPARE'
call checkerr (sqlca, errloc, *999)

EXEC SQL DECLARE c1 CURSOR FOR s1 2

parm_var = 'Mgr'
EXEC SQL OPEN c1 USING :parm_var 3
errloc = 'OPEN'
call checkerr (sqlca, errloc, *999)

parm_var = 'Clerk'
st = 'UPDATE staff SET job = ? WHERE CURRENT OF c1'
EXEC SQL PREPARE s2 FROM :st 4

10 continue
  EXEC SQL FETCH c1 INTO :pname, :dept 5
  if (sqlcode .ne. 0) goto 100
```

```

        print *, pname, 'in dept. ', dept, ' will be demoted to Clerk'

        EXEC SQL EXECUTE s2 USING :parm_var 6
        goto 10

100 EXEC SQL CLOSE c1 7
    errloc = 'CLOSE'
    call checkerr (sqlca, errloc, *999)

    EXEC SQL ROLLBACK
    errloc = 'ROLLBACK'
    call checkerr (sqlca, errloc, *999)
    print *, 'On second thought -- changes rolled back.'

    EXEC SQL CONNECT RESET
    errloc = 'CONNECT RESET'
    call checkerr (sqlca, errloc, *999)

999 stop
end

```

The DB2 Call Level Interface (CLI)

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code, which is then compiled, bound to the database, and executed. In contrast, a DB2 CLI application does not have to be precompiled or bound, but instead uses a standard set of functions to execute SQL statements and related services at run time.

This difference is important because, traditionally, precompilers have been specific to each database product, which effectively ties your applications to that product. DB2 CLI enables you to write portable applications that are independent of any particular database product. This independence means DB2 CLI applications do not have to be recompiled or rebound to access different DB2 databases, including DRDA databases. They just connect to the appropriate database at run time.

Comparing Embedded SQL and DB2 CLI

DB2 CLI and embedded SQL also differ in the following ways:

- DB2 CLI does not require the explicit declaration of cursors. DB2 CLI has a supply of cursors that get used as needed. The application can then use the generated cursor in the normal cursor fetch model for multiple row SELECT statements and positioned UPDATE and DELETE statements.
- The OPEN statement is not used in DB2 CLI. Instead, the execution of a SELECT automatically causes a cursor to be opened.
- Unlike embedded SQL, DB2 CLI allows the use of parameter markers on the equivalent of the EXECUTE IMMEDIATE statement (the `SQLExecDirect()` function).
- A COMMIT or ROLLBACK in DB2 CLI is issued via the `SQLEndTran()` function call rather than by passing it as an SQL statement.
- DB2 CLI manages statement related information on behalf of the application, and provides a *statement handle* to refer to it as an abstract object. This handle eliminates the need for the application to use product specific data structures.
- Similar to the statement handle, the *environment handle* and *connection handle* provide a means to refer to all global variables and connection specific information. The *descriptor handle* describes either the parameters of an SQL statement or the columns of a result set.
- DB2 CLI uses the SQLSTATE values defined by the X/Open SQL CAE specification. Although the format and most of the values are consistent with values used by the IBM relational database products, there are differences. (There are also differences between ODBC SQLSTATES and the X/Open defined SQLSTATES).

- DB2 CLI supports scrollable cursors. With scrollable cursors, you can scroll through a static cursor as follows:
 - Forward by one or more rows
 - Backward by one or more rows
 - From the first row by one or more rows
 - From the last row by one or more rows.

Despite these differences, there is an important common concept between embedded SQL and DB2 CLI: *DB2 CLI can execute any SQL statement that can be prepared dynamically in embedded SQL.*

Note: DB2 CLI can also accept some SQL statements that cannot be prepared dynamically, such as compound SQL statements.

Table 26 on page 629 lists each SQL statement, and indicates whether or not it can be executed using DB2 CLI. The table also indicates if the command line processor can be used to execute the statement interactively, (useful for prototyping SQL statements).

Each DBMS may have additional statements that you can dynamically prepare. In this case, DB2 CLI passes the statements to the DBMS. There is one exception: the COMMIT and ROLLBACK statement can be dynamically prepared by some DBMSs but are not passed. In this case, use the `SQLEndTran()` function to specify either the COMMIT or ROLLBACK statement.

Advantages of Using DB2 CLI

The DB2 CLI interface has several key advantages over embedded SQL.

- It is ideally suited for a client-server environment, in which the target database is not known when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application is connected to.
- It increases the portability of applications by removing the dependence on precompilers.
- Individual DB2 CLI applications do not need to be bound to each database, only bind files shipped with DB2 CLI need to be bound once for all DB2 CLI applications. This can significantly reduce the amount of management required for the application once it is in general use.
- DB2 CLI applications can connect to multiple databases, including multiple connections to the same database, all from the same application. Each connection has its own commit scope. This is much simpler using CLI than using embedded SQL where the application must make use of multi-threading to achieve the same result.

- DB2 CLI eliminates the need for application controlled, often complex data areas, such as the SQLDA and SQLCA, typically associated with embedded SQL applications. Instead, DB2 CLI allocates and controls the necessary data structures, and provides a *handle* for the application to reference them.
- DB2 CLI enables the development of multi-threaded thread-safe applications where each thread can have its own connection and a separate commit scope from the rest. DB2 CLI achieves this by eliminating the data areas described above, and associating all such data structures that are accessible to the application with a specific handle. Unlike embedded SQL, a multi-threaded CLI application does not need to call any of the context management DB2 APIs; this is handled by the DB2 CLI driver automatically.
- DB2 CLI provides enhanced parameter input and fetching capability, allowing arrays of data to be specified on input, retrieving multiple rows of a result set directly into an array, and executing statements that generate multiple result sets.
- DB2 CLI provides a consistent interface to query catalog (Tables, Columns, Foreign Keys, Primary Keys, etc.) information contained in the various DBMS catalog tables. The result sets returned are consistent across DBMSs. This shields the application from catalog changes across releases of database servers, as well as catalog differences amongst different database servers; thereby saving applications from writing version specific and server specific catalog queries.
- Extended data conversion is also provided by DB2 CLI, requiring less application code when converting information between various SQL and C data types.
- DB2 CLI incorporates both the ODBC and X/Open CLI functions, both of which are accepted industry specifications. DB2 CLI is also aligned with the emerging ISO CLI standard. Knowledge that application developers invest in these specifications can be applied directly to DB2 CLI development, and vice versa. This interface is intuitive to grasp for those programmers who are familiar with function libraries but know little about product specific methods of embedding SQL statements into a host language.
- DB2 CLI provides the ability to retrieve multiple rows and result sets generated from a stored procedure residing on a DB2 Universal Database (or DB2 for MVS/ESA version 5 or later) server. However, note that this capability exists for Version 5 DB2 Universal Database clients using embedded SQL if the stored procedure resides on server that is accessible from a DataJoiner Version 2 server.
- DB2 CLI supports server-side scrollable cursors that can be used in conjunction with array output. This is useful in GUI applications that display database information in scroll boxes that make use of the Page Up, Page Down, Home and End keys. You can declare a read-only cursor as

scrollable then move forward or backward through the result set by one or more rows. You can also fetch rows by specifying an offset from:

- The current row
 - The beginning or end of the result set
 - A specific row you have previously set with a bookmark.
- DB2 CLI applications can dynamically describe parameters in an SQL statement the same way that CLI and Embedded SQL applications describe result sets. This enables CLI applications to dynamically process SQL statements that contain parameter markers without knowing the data type of those parameter markers in advance. When the SQL statement is prepared, describe information is returned detailing the data types of the parameters.

Deciding on Embedded SQL or DB2 CLI

Which interface you choose depends on your application.

DB2 CLI is ideally suited for query-based graphical user interface (GUI) applications that require portability. The advantages listed above, may make using DB2 CLI seem like the obvious choice for any application. There is however, one factor that must be considered, the comparison between static and dynamic SQL. It is much easier to use static SQL in embedded applications.

For more information on using static SQL in CLI applications, refer to the Web page at:

<http://www.software.ibm.com/data/db2/udb/staticcli>

Static SQL has several advantages:

- Performance

Dynamic SQL is prepared at run time, static SQL is prepared at precompile time. As well as requiring more processing, the preparation step may incur additional network-traffic at run time. This additional step (and network-traffic), however, will not be required if the DB2 CLI application makes use of deferred prepare.

It is important to note that static SQL will not always have better performance than dynamic SQL. Dynamic SQL can make use of changes to the database, such as new indexes, and can use current database statistics to choose the optimal access plan. In addition, precompilation of statements can be avoided if they are cached.

- Encapsulation and Security

In static SQL, the authorizations to objects (such as a table, view) are associated with a package and are validated at package binding time. This means that database administrators need only to grant execute on a

particular package to a set of users (thus encapsulating their privileges in the package) without having to grant them explicit access to each database object. In dynamic SQL, the authorizations are validated at run time on a per statement basis; therefore, users must be granted explicit access to each database object. This permits these users access to parts of the object that they do not have a need to access.

- Embedded SQL is supported in languages other than C or C++.
- For fixed query selects, embedded SQL is simpler.

If an application requires the advantages of both interfaces, it is possible to make use of static SQL within a DB2 CLI application by creating a stored procedure that contains the static SQL. The stored procedure is called from within a DB2 CLI application and is executed on the server. Once the stored procedure is created, any DB2 CLI or ODBC application can call it. For more information, refer to the *CLI Guide and Reference*.

For more information on using static SQL in CLI applications, refer to the Web page at:

<http://www.software.ibm.com/data/db2/udb/staticcli>

It is also possible to write a mixed application that uses both DB2 CLI and embedded SQL, taking advantage of their respective benefits. In this case, DB2 CLI is used to provide the base application, with key modules written using static SQL for performance or security reasons. This complicates the application design, and should only be used if stored procedures do not meet the applications requirements. For more information, refer to the section on *Mixing Embedded SQL and DB2 CLI* in the *CLI Guide and Reference*.

Ultimately, the decision on when to use each interface, will be based on individual preferences and previous experience rather than on any one factor.

Part 3. Stored Procedures

Chapter 6. Stored Procedures

Stored Procedure Overview	185	Code Page Considerations.	208
Advantages of Stored Procedures	186	C++ Consideration	208
Writing Stored Procedures.	188	Graphic Host Variable Considerations	209
Client Application	189	Multisite Update Consideration	209
Allocating Host Variables	189	Building the Stored Procedure	209
Calling Stored Procedures.	190	Client Application	210
Running the Client Application	190	Stored Procedure.	210
Stored Procedure on the Server	190	NOT FENCED Stored Procedures	210
Registering Stored Procedures	190	Returning Result Sets from Stored	
Variable Declaration and CREATE		Procedures	212
PROCEDURE Examples	196	Resolving Problems	214
SQL Statements in Stored Procedures	197	Overview of Stored Procedure Builder	215
Restrictions.	198	What is SPB?	215
Return Values.	199	Advantages of Using SPB	216
Example OUT Stored Procedure.	200	Creating New Stored Procedures	216
OUT Client Description	202	Working with Existing Stored	
Example Client Application:		Procedures	217
Outcli.java	204	Creating SPB Projects	217
OUT Stored Procedure Description	206	Debugging Stored Procedures	217
Example Stored Procedure: Outsrv.sqlj	207		

Stored Procedure Overview

You can use a *stored procedure*, a procedure stored on a database server that executes and accesses the database locally, to return information to client applications. A stored procedure saves the overhead of having a remote application pass multiple SQL commands to a database on a server. With a single statement, a client application can call the stored procedure, which then performs the database access work and returns the results to the client application.

To create a stored procedure, you must write the application in two separate procedures. The calling procedure is contained in a *client application* and executes on the client. The stored procedure executes at the location of the database on the database server.

You can write stored procedures in any language supported by DB2 on your operating system. You do not have to write client applications in the same language as the stored procedure. DB2 transparently passes the values between client application and stored procedure.

You can use the DB2 Stored Procedure Builder (SPB) to help develop Java stored procedures. SPB can be integrated with popular application

development tools, including Microsoft Visual Studio and IBM Visual Age for Java, or you can use it as a standalone utility. To help you create your stored procedures, SPB provides design assistants that guide you through basic design patterns, help you create SQL queries, and estimate the performance cost of invoking a stored procedure.

For more information on the DB2 Stored Procedure Builder, please see “Overview of Stored Procedure Builder” on page 215.

Advantages of Stored Procedures

Figure 3 shows how a normal database manager application accesses a database located on a database server.

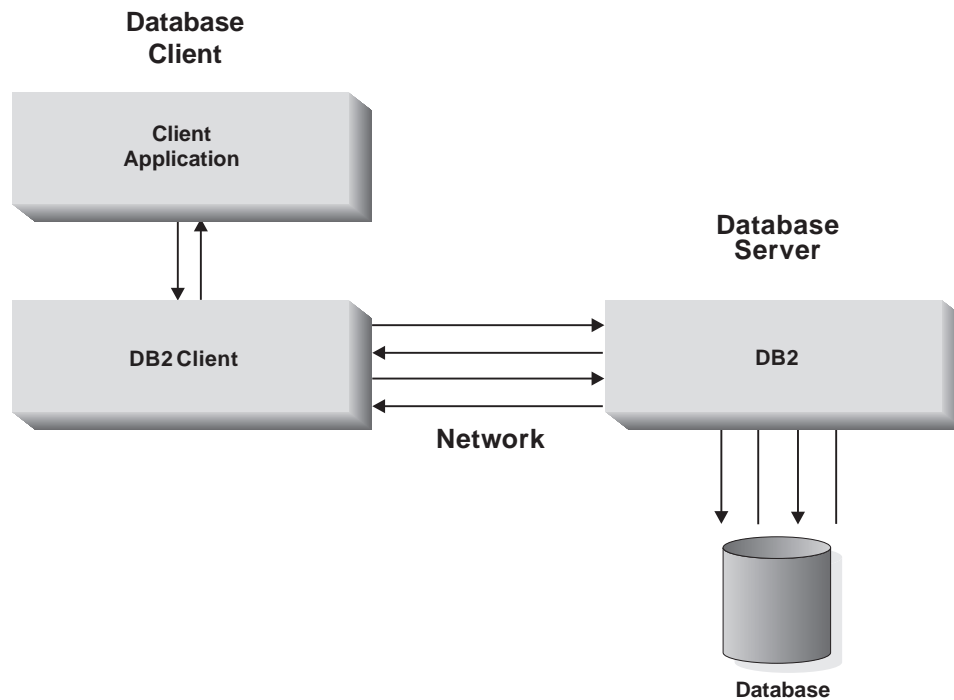


Figure 3. Application Accessing a Database on a Server

All database access must go across the network which, in some cases, results in poor performance.

Using stored procedures allows a client application to pass control to a stored procedure on the database server. This allows the stored procedure to perform intermediate processing on the database server, *without transmitting*

unnecessary data across the network. Only those records that are actually required at the client need to be transmitted. This can result in reduced network traffic and better overall performance. Figure 4 shows this feature.

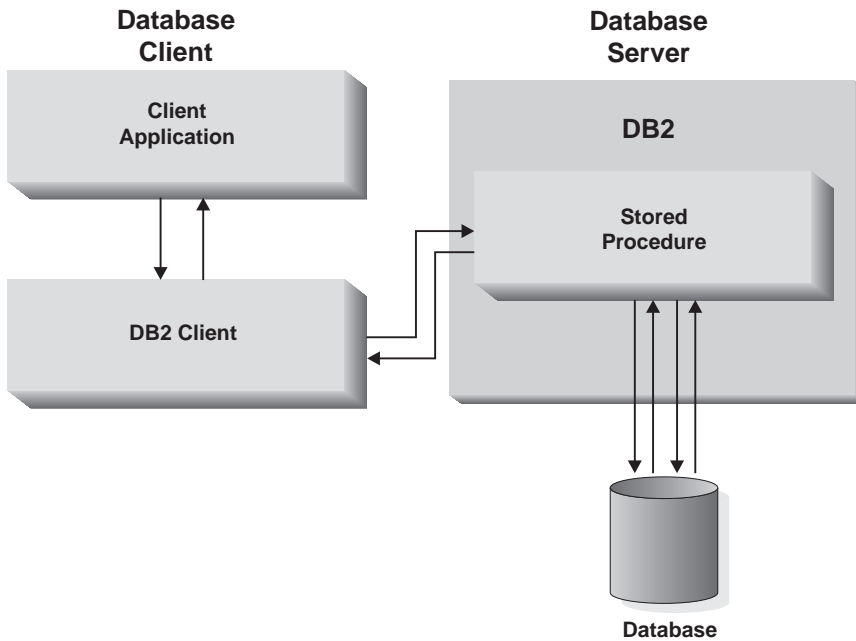


Figure 4. Application Using a Stored Procedure

Applications using stored procedures have the following advantages:

- Reduced network traffic
A properly designed application that processes large amounts of data using stored procedures returns only the data that is needed by the client. This reduces the amount of data transmitted across the network.
- Improved performance of server intensive work
The more SQL statements that are grouped together for execution, the larger the savings in network traffic. A typical application requires two trips across the network for each SQL statement, whereas an application using the stored procedure technique requires two trips across the network for each *group* of SQL statements. This reduces the number of trips, resulting in a savings from the overhead associated with each trip.
- Access to features that exist only on the database server, including:
 - Commands to list directories on the server (such as LIST DATABASE DIRECTORY and LIST NODE DIRECTORY) can only run on the server.
 - The stored procedure may have the advantage of increased memory and disk space if the server computer is so equipped.

- Additional software installed only on the database server could be accessed by the stored procedure.

Writing Stored Procedures

An application using stored procedures must be written in two separate procedures. The *calling procedure* is contained in a client application, and executes on the client. It can be written in any of the supported host languages. The *stored procedure* executes at the location of the database on the database server, and must be written in one of the supported languages for that database server.

The two procedures must be built in separate steps.

The client application performs the following:

1. Declares, allocates, and initializes storage for the optional data structures and host variables.
2. Connects to a database by executing the CONNECT TO statement, or by doing an implicit connect. Refer to the *SQL Reference* for details.
3. Invokes the stored procedure through the SQL CALL statement.
4. Issues a COMMIT or ROLLBACK to the database.
5. Disconnects from the database.

Note that you can code SQL statements in any of the above steps.

When invoked, the stored procedure performs the following:

1. Accepts the parameters from the client application.
2. Executes on the database server under the same transaction as the client application.
3. Returns SQLCA information and optional output data to the client application.

The stored procedure executes when called by the client application. Control is returned to the client when the server procedure finishes processing. You can put several stored procedures into one library.

This chapter describes how to write stored procedures with the following parameter styles:

GENERAL The stored procedure receives parameters as host variables from the CALL statement in the client application. The stored procedure does **not** directly pass null indicators to the client application. GENERAL is the equivalent of SIMPLE stored procedures for DB2 Universal Database for OS/390.

GENERAL WITH NULLS

The stored procedure receives a vector of null indicators for each of the parameters on the CALL. Like GENERAL, parameters are passed as host variables. GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS stored procedures for DB2 Universal Database for OS/390.

JAVA

The stored procedure uses a parameter passing convention that conforms to the **SQLJ Routines** specification. The stored procedure receives IN parameters as host variables, and receives OUT and INOUT parameters as single entry arrays.

You **must** register each stored procedure for the previously listed parameter styles with a CREATE PROCEDURE statement. The CREATE PROCEDURE statement specifies the procedure name, arguments, location, and parameter style of each stored procedure. These parameter styles offer increased portability and scalability of your stored procedure code across the DB2 family.

For information on using the styles of stored procedures supported by versions of DB2 prior to Version 6, that is, the DB2DARI and DB2GENERAL parameter styles, see “Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs” on page 655.

Client Application

The client application performs several steps before calling the stored procedure. It must be connected to a database, and it must declare, allocate, and initialize host variables or SQLDA structure. The SQL CALL statement can accept a series of host variables, or an SQLDA structure. Refer to the *SQL Reference* for descriptions of the SQL CALL statement and the SQLDA structure. For information on using the SQLDA structure in a client application, see “Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs” on page 655.

Allocating Host Variables

Use the following steps to allocate the necessary input host variables on the client side of a stored procedure:

1. Declare enough host variables for all input variables that will be passed to the stored procedure.
2. Determine which input host variables can also be used to return values back from the stored procedure to the client.
3. Declare host variables for any additional values returned from the stored procedure to the client.

When writing the client portion of your stored procedure, you should attempt to overload as many of the host variables as possible by using them for both input and output. This will increase the efficiency of handling multiple host variables. For example, when returning an SQLCODE to the client from the stored procedure, try to use an input host variable that is declared as an INTEGER to return the SQLCODE.

Note: Do not allocate storage for these structures on the database server. The database manager automatically allocates duplicate storage based upon the storage allocated by the client application. Do not alter any storage pointers for the input/output parameters on the stored procedure side. Attempting to replace a pointer with a locally created storage pointer will cause an error with SQLCODE -1133 (SQLSTATE 39502).

Calling Stored Procedures

You can invoke a stored procedure stored at the location of the database by using the SQL CALL statement. Refer to the *SQL Reference* for a complete description of the CALL statement. Using the CALL statement is the recommended method of invoking stored procedures.

Running the Client Application

The client application must ensure that a database connection has been made before invoking the stored procedure, or an error is returned. After the database connection and data structure initialization, the client application calls the stored procedure and passes any required data. The application disconnects from the database. Note that you can code SQL statements in any of the above steps.

Stored Procedure on the Server

The stored procedure is invoked by the SQL CALL statement and executes using data passed to it by the client application. The parameter style with which you register the stored procedure in the database manager with the CREATE PROCEDURE statement determines how the stored procedure receives data from the client application.

Registering Stored Procedures

To use the CREATE PROCEDURE statement, you **must** declare the following:

- Procedure name
- Mode, name, and SQL data type of each parameter
- EXTERNAL name and location
- PARAMETER STYLE

Your CREATE PROCEDURE **should** also declare the following:

- Whether it runs FENCED or NOT FENCED
- The type of SQL statements contained in the procedure body, if any

You can find more information on the CREATE PROCEDURE statement, including its full syntax and options for DB2 family compatibility, in the *SQL Reference*. Descriptions of typical usages of the CREATE PROCEDURE statement follow.

Procedure Names: You can overload stored procedures only by using the same name for procedures that accept a unique number of parameters. Since DB2 does not distinguish between data types, you cannot overload stored procedures based on parameter data types.

For example, issuing the following CREATE PROCEDURE statements will work because they accept one and two parameters, respectively:

```
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER) ...  
CREATE PROCEDURE OVERLOAD (IN VAR1 INTEGER, IN VAR2 INTEGER) ...
```

However, DB2 will fail to register the second stored procedure in the following example because it has the same number of parameters as the first stored procedure with the same name:

```
CREATE PROCEDURE OVERLOADFAIL (IN VAR1 INTEGER) ...  
CREATE PROCEDURE OVERLOADFAIL (IN VAR2 VARCHAR(15)) ...
```

Parameter Modes: You must declare every parameter passed from a client application to the stored procedure, and from the stored procedure back to the client application, as either an IN, OUT, or INOUT parameter. For each parameter, you must also specify a name and SQL data type. See “Variable Declaration and CREATE PROCEDURE Examples” on page 196 for examples.

IN Passes a value to the stored procedure from the client application, but returns no value to the client application when control returns to the client application

OUT Stores a value that is passed to the client application when the stored procedure terminates

INOUT Passes a value to the stored procedure from the client application, and returns a value to the client application when the stored procedure terminates

Location: The EXTERNAL clause of the CREATE PROCEDURE statement tells the database manager the location of the library that contains the stored procedure. If you do not specify an absolute path for the library, or a jar name

for Java stored procedures, the database manager searches the *function directory*. The *function directory* is a directory defined for your operating system as follows:

Unix operating systems

sqllib/function

OS/2 or Windows 32-bit operating systems

instance_name\function, where *instance_name* represents the value of the *DB2INSTPROF* instance-specific registry setting. If DB2 does not find the stored procedure in *instance_name*\function, DB2 searches the directories defined by the *PATH* and *LIBPATH* environment variables.

For example, the function directory for a Windows NT server with DB2 installed in the C:\sqllib directory, and with no specified *DB2INSTPROF* registry setting, is:

C:\sqllib\function

Note: You should give your library a name that is different than the stored procedure name. If DB2 locates the library in the search path, DB2 executes a stored procedure with the same name as its library as a FENCED DB2DARI procedure.

For LANGUAGE C stored procedures, specify:

- The library name, taking the form of either:
 - A library found in the function directory
 - An absolute path including the library name
- The entry point for the stored procedure in the library. If you do not specify an entry point, the database manager will use the default entry point. The IBM XL C compiler on AIX allows you to specify any exported function name in the library as the default entry point. This is the function that is called if only the library name is specified in a stored procedure call or CREATE FUNCTION statement. To specify a default entry point, use the -e option in the link step. For example: -e *funcname* makes *funcname* the default entry point. On other UNIX platforms, no such mechanism exists, so the default entry point is assumed by DB2 to be the same name as the library itself.

On a UNIX-based system, for example, mymod!proc8 directs the database manager to the sqllib/function/mymod library and to use entry point proc8 within that library. On Windows 32-bit and OS/2 operating systems mymod!proc8 directs the database manager to load the mymod.dll file from the function directory and call the proc8() procedure in the dynamic link library (DLL).

For LANGUAGE JAVA stored procedures, use:

- (Option): The name you gave to the jar file when you installed it in the database, followed by a colon :. You can qualify the jar name with a schema name. For more information on installing jar files, see “Java Stored Procedures and UDFs” on page 562.

If you do not specify a jar file, the database manager looks for the class in the function directory.

- The class name; if the class is part of a package, include the complete package prefix.
- The method name

For example, if you specify `MyPackage.MyClass.myMethod`, the database manager uses the `myMethod` method in the `MyClass` class, within the `MyPackage` package. Since you did not specify a JAR file, the database manager will search the function directory.

For more information on the function directory, see “Location” on page 191.

LANGUAGE: For C/C++, declare LANGUAGE C in your CREATE PROCEDURE statement. For Java stored procedures, declare LANGUAGE JAVA. For COBOL stored procedures, declare LANGUAGE COBOL. For Fortran or REXX stored procedures, you must write the stored procedure as a DB2DARI stored procedure. For more information on writing DB2DARI stored procedures, see “Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs” on page 655.

LANGUAGE C

The database manager calls the stored procedure using ANSI C calling and linkage conventions. Use this option for most C/C++ stored procedures.

LANGUAGE JAVA

The database manager calls the stored procedure as a method in a Java class. Use this option for any Java stored procedure.

LANGUAGE COBOL

The database manager calls the stored procedure using COBOL calling and linkage conventions. Use this option for COBOL stored procedures.

PARAMETER STYLES: Table 9 on page 194 summarizes the combinations of PARAMETER STYLE (horizontal axis) and LANGUAGE (vertical axis) allowed in CREATE PROCEDURE statements for DB2 Version 6.

Table 9. CREATE PROCEDURE: Valid Combinations of PARAMETER STYLE and LANGUAGE

	GENERAL, GENERAL WITH NULLS	JAVA	DB2SQL	DB2DARI	DB2GENERAL
LANGUAGE C	Y	N	Y	Y	N
LANGUAGE JAVA	N	Y	N	N	Y
LANGUAGE COBOL	Y	N	Y	N	N

GENERAL

The stored procedure receives parameters as host variables from the CALL statement in the client application. The stored procedure does **not** directly pass null indicators to the client application. You can only use GENERAL when you also specify the LANGUAGE C or LANGUAGE COBOL option.

DB2 Universal Database for OS/390 compatibility note: GENERAL is the equivalent of SIMPLE.

GENERAL WITH NULLS

The stored procedure receives a vector of null indicators for each of the parameters on the CALL. Like GENERAL, parameters are passed as host variables. You can only use GENERAL WITH NULLS when you also specify the LANGUAGE C or LANGUAGE COBOL option.

DB2 Universal Database for OS/390 compatibility note: GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS

JAVA The stored procedure uses a parameter passing convention that conforms to the *SQLJ Routines* specification. The stored procedure receives IN parameters as host variables, and receives OUT and INOUT parameters as single entry arrays. You can only use JAVA when you also specify the LANGUAGE JAVA option.

DB2SQL

DB2 passes the following arguments to the stored procedure:

- Parameters from the CALL statement as host variables
- A NULL indicator for each parameter
- An SQLSTATE variable to return to DB2
- The qualified name of the stored procedure
- The specific name of the stored procedure
- An SQL diagnostic string to return to DB2

You can only specify DB2SQL when you also specify the LANGUAGE C or LANGUAGE COBOL option.

DB2GENERAL

The stored procedure uses a parameter passing convention that is only supported by DB2 Java stored procedures. You can only use DB2GENERAL when you also specify the LANGUAGE JAVA option.

For increased portability, you should write Java stored procedures using the PARAMETER STYLE JAVA conventions. See “Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs” on page 655 for more information on writing DB2GENERAL parameter style stored procedures.

DB2DARI

The stored procedure uses a parameter passing convention that conforms with C language calling and linkage conventions. This option is only supported by DB2 Universal Database, and can only be used when you also specify the LANGUAGE C option.

To increase portability across the DB2 family, you should write your LANGUAGE C stored procedures using the GENERAL or GENERAL WITH NULLS parameter styles. If you want to write DB2DARI parameter style stored procedures, see “Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs” on page 655.

Passing a DBINFO Structure: For LANGUAGE C stored procedures with a PARAMETER TYPE of GENERAL, GENERAL WITH NULLS, or DB2SQL, you have the option of writing your stored procedure to accept an additional parameter. You can specify DBINFO in the CREATE PROCEDURE statement to instruct the client application to pass a DBINFO structure containing information about the DB2 client to the stored procedure, along with the call parameters. The DBINFO structure contains the following values:

Data base name

The name of the database to which the client is connected.

Application authorization ID

The application run-time authorization ID.

Code page

The code page of the database.

Schema name

Not applicable to stored procedures.

Table name

Not applicable to stored procedures.

Column name

Not applicable to stored procedures.

Database version and release

The version, release, and modification level of the database server invoking the stored procedure.

Platform

The platform of the database server.

Table function result column numbers

Not applicable to stored procedures.

For more information on the DBINFO structure, see “DBINFO Structure” on page 297.

Writing Stored Procedures as main Functions: For LANGUAGE C stored procedures with a PARAMETER TYPE of GENERAL, GENERAL WITH NULLS, or DB2SQL, you have the option of writing your stored procedure to accept parameters like a main function in a C program (MAIN) or like a subroutine (SUB). The default value for PROGRAM TYPE is SUB, which accepts parameters as separate arguments; for example:

```
void storproc (int arg1, long arg2, int arg3);
```

To write a stored procedure that accepts arguments like a main function in a C program, specify PROGRAM TYPE MAIN in the CREATE PROCEDURE statement. You must write stored procedures of PROGRAM TYPE MAIN to conform to the following specifications:

- the function name must be `main`
- the stored procedure accepts parameters through two arguments:
 - a parameter counter variable; for example, `argc`
 - an array containing the parameters; for example, `argv[]`
- the stored procedure must be built as a shared library

For example, the signature for a stored procedure of PROGRAM TYPE MAIN would appear as follows:

```
int main(int argc, char* argv[])
```

Variable Declaration and CREATE PROCEDURE Examples

The following examples demonstrate the stored procedure source code and CREATE PROCEDURE statements you would use in hypothetical scenarios with the SAMPLE database.

Using IN and OUT Parameters: Assume that you want to create a Java stored procedure GET_LASTNAME that, given EMPNO (SQL type VARCHAR), returns LASTNAME (SQL type CHAR) from the EMPLOYEE table in the SAMPLE database. You will create the procedure as the getname

method of the Java class `StoredProcedure`, contained in the JAR installed as `myJar`. Finally, you will call the stored procedure with a client application coded in C.

1. Declare two host variables in your stored procedure source code:

```
String empid;
String name;
...
#sql { SELECT lastname INTO :empid FROM employee WHERE empno=:empid }
```

2. Register the stored procedure with the following `CREATE PROCEDURE` statement:

```
CREATE PROCEDURE GET_LASTNAME (IN EMPID CHAR(6), OUT NAME VARCHAR(15))
EXTERNAL NAME 'myJar:StoredProcedure.getname'
LANGUAGE JAVA PARAMETER STYLE JAVA FENCED
READS SQL DATA
```

3. Call the stored procedure from your client application written in C:

```
EXEC SQL BEGIN DECLARE SECTION;
    struct name { short int; char[15] }
    char[7] empid;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CALL GET_LASTNAME (:empid, :name);
```

Using INOUT Parameters: For the following example, assume that you want to create a C stored procedure `GET_MANAGER` that, given `DEPTNUMB` (SQL type `SMALLINT`), returns `MANAGER` (SQL type `SMALLINT`) from the `ORG` table in the `SAMPLE` database.

1. Since `DEPTNUMB` and `MANAGER` are both of SQL data type `SMALLINT`, you can declare a single variable in your stored procedure that receives a value from and returns a value to the client application:

```
EXEC SQL BEGIN DECLARE SECTION;
    short onevar = 0;
EXEC SQL END DECLARE SECTION;
```

2. Register the stored procedure with the following `CREATE PROCEDURE` statement:

```
CREATE PROCEDURE GET_MANAGER (INOUT ONEVAR SMALLINT)
EXTERNAL NAME 'stplib!getman'
LANGUAGE C PARAMETER STYLE GENERAL FENCED
READS SQL DATA
```

3. Call the stored procedure from your client application written in Java:

```
short onevar = 0;
...
#SQL { CALL GET_MANAGER (:INOUT onevar) };
```

SQL Statements in Stored Procedures

Stored procedures can contain SQL statements. When you issue the `CREATE PROCEDURE` statement, you should specify the type of SQL statements the

stored procedure contains, if any. If you do not specify a value when you register the stored procedure, the database manager uses the restrictive default value **CONTAINS SQL**. To indicate the type of SQL used in the stored procedure, you can use one of the following four options:

NO SQL

No SQL statements are allowed in the stored procedure. The database manager returns an error if the stored procedure attempts to execute any SQL statement.

CONTAINS SQL

The database manager returns an error if the stored procedure attempts to execute any of the following SQL statements:

- COMMIT
- DELETE
- DESCRIBE
- INSERT
- PREPARE INTO
- ROLLBACK
- SELECT
- UPDATE

READS SQL DATA

The database manager returns an error if the stored procedure attempts to execute any of the following SQL statements:

- COMMIT
- DELETE
- INSERT
- ROLLBACK
- UPDATE

MODIFIES SQL DATA

The database manager returns an error if the stored procedure attempts to execute any of the following SQL statements:

- COMMIT
- ROLLBACK

For more information on the **CREATE PROCEDURE** statement, refer to the *SQL Reference*.

Restrictions

When you create a stored procedure, you must observe the following restrictions:

- Do not use the standard I/O streams, for example, calls to `System.out.println()` or `printf()`. Stored procedures run in the background, so you cannot write to the screen. However, you can write to a file.
- Include only the SQL statements allowed by the `CREATE PROCEDURE` statement with which you register the stored procedure. See “SQL Statements in Stored Procedures” on page 197 for the effects of `CONTAINS SQL`, `NO SQL`, `READS SQL DATA`, `MODIFIES SQL DATA`.
- `COMMIT` statements are **not** allowed in any registered stored procedure; a `COMMIT` statement can only be issued by the client application.

Note: For compatibility with previous versions of DB2, DB2DARI and DB2GENERAL stored procedures allow `COMMIT` statements within the procedure body.

- You cannot execute any connection-related statements or commands, such as:
 - `BACKUP`
 - `CONNECT`
 - `CONNECT TO`
 - `CONNECT RESET`
 - `CREATE DATABASE`
 - `DROP DATABASE`
 - `FORWARD RECOVERY`
 - `RESTORE`

in a stored procedure.

- On UNIX-based systems, `NOT FENCED` stored procedures run under the `userid` of the DB2 Agent Process. `FENCED` stored procedures run under the `userid` that owns the `db2dari` executable. This `userid` controls the system resources available to the stored procedure. For information on the `db2dari` executable, refer to the *Quick Beginnings* book for your platform.
- You cannot overload stored procedures that accept the same number of parameters, even if the parameters are of different SQL data types.
- Stored procedures cannot contain commands that would terminate the current process. A stored procedure should always return control to the client without terminating the current process.

Return Values

The return value of the stored procedure is never returned to the client application. It is used by the database manager to determine if the server procedure should be released from memory upon exit.

Note: Java stored procedures do not return values to the database manager. This section applies only to stored procedures that are not written in Java.

The stored procedure returns one of the following values:

SQLZ_DISCONNECT_PROC

Tells the database manager to release (unload) the library.

SQLZ_HOLD_PROC

Tells the database manager to keep the server library in main memory so that the library will be ready for the next invocation of the stored procedure. This may improve performance.

If the stored procedure is invoked only once, it should return SQLZ_DISCONNECT_PROC.

If the client application issues multiple calls to invoke the same stored procedure, SQLZ_HOLD_PROC should be the return value of the stored procedure. The stored procedure will not be unloaded.

If SQLZ_HOLD_PROC is used, the last invocation of the stored procedure should return the value SQLZ_DISCONNECT_PROC to free the stored procedure library from main memory. Otherwise, the library remains in main memory until the database manager is stopped. As an alert to the stored procedure, the client application could pass a flag in one of the parameters indicating the final call.

Example OUT Stored Procedure

Following is a sample program demonstrating the use of an OUT host variable. The client application invokes a stored procedure that determines the median salary for employees in the SAMPLE database. (The definition of the median is that half the values lie above it, and half below it.) The median salary is then passed back to the client application using an OUT host variable.

This sample program calculates the median salary of all employees in the SAMPLE database. Since there is no existing SQL column function to calculate medians, the median salary can be found iteratively by the following algorithm:

1. Determine the number of records, n , in the table.
2. Order the records based upon salary.
3. Fetch records until the record in row position $n/2+1$ is found.
4. Read the median salary from this record.

An application that uses neither the stored procedures technique, nor blocking cursors, must FETCH each salary across the network as shown in Figure 5.

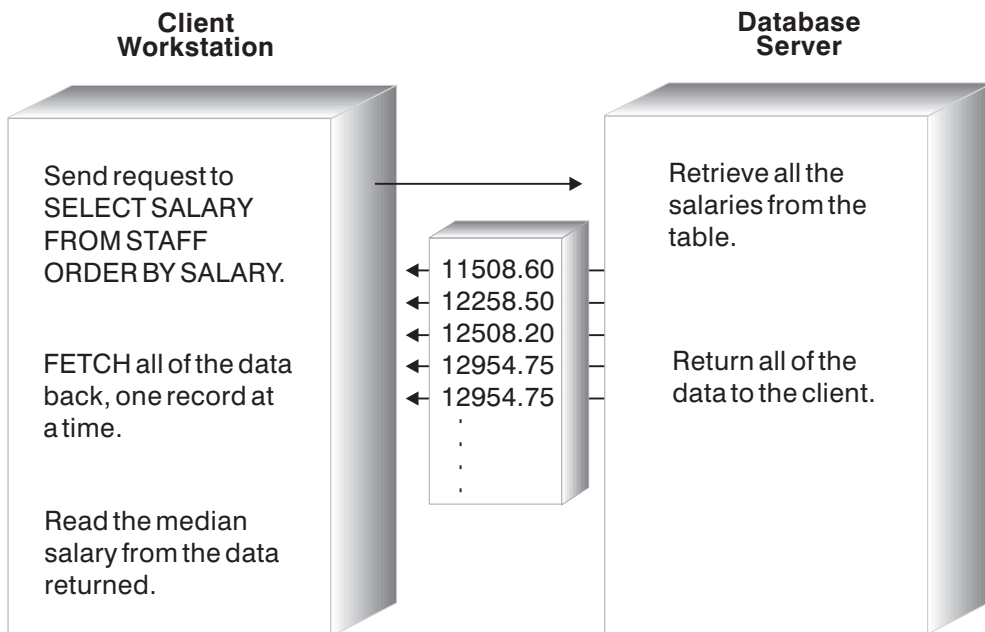


Figure 5. Median Sample Without a Stored Procedure

Since only the salary at row $n/2+1$ is needed, the application discards all the additional data, *but only after it is transmitted across the network*.

You can design an application using the stored procedures technique that allows the stored procedure to process and discard the unnecessary data, returning only the median salary to the client application. Figure 6 shows this feature.

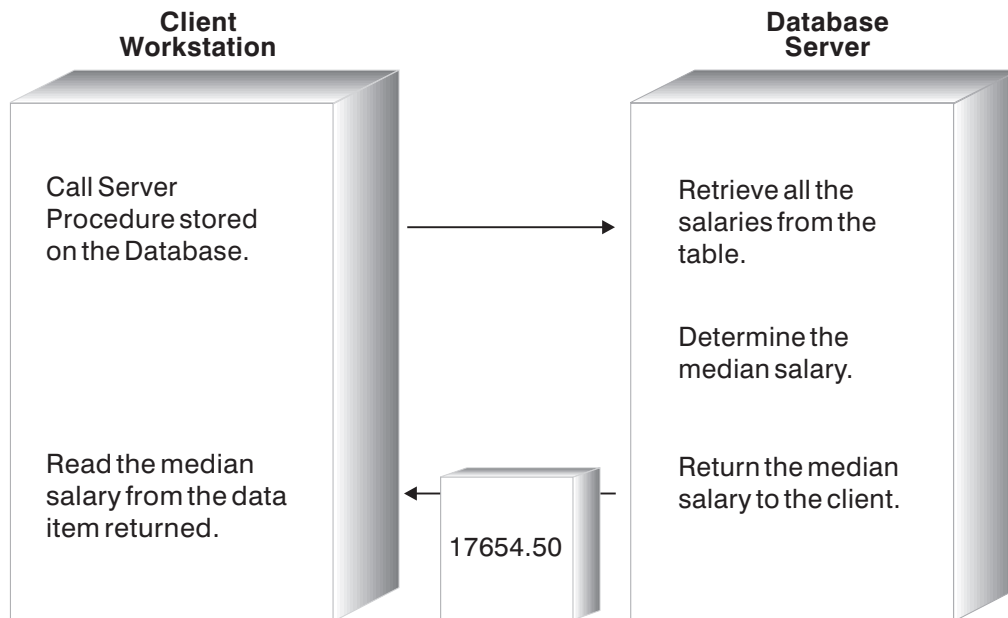


Figure 6. Median Sample Using a Stored Procedure

“OUT Client Description” shows a sample OUT host variable client application and stored procedure. The sample programs are available in Java as:

Client application	Outcli.java
Stored procedure	Outsrv.sqlj

For the OUT host variable client application and stored procedure in other programming languages, refer to the `sqllib/samples` directory.

OUT Client Description

1. **Include Files.** The program begins with the following include files:

SQL	Defines the symbol <code>SQL_TYP_FLOAT</code>
SQLDA	Defines the descriptor area
SQLCA	Defines the communication area for error handling
2. **Connect to Database.** The application must connect to the database before registering or invoking the stored procedure.
3. **Register the Stored Procedure.** The stored procedure is normally registered outside of the client application, but is included in the client application to demonstrate how to register a stored procedure. The parameter list contains one output (OUT) host variable of type `DOUBLE`.

4. **Declare and Initialize the Host Variable.** This step declares and initializes the host variable. Java programs must register the data type of each parameter before invoking the stored procedure.
5. **Invoke the Stored Procedure.** The application invokes the procedure `outsrv` at the location of the database, `sample`, using a `CALL` statement with one host variable.
6. **Disconnect from Database.** A Java application automatically disconnects from the database when the application terminates.

The `CHECKERR` macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C	<code>check_error</code> is redefined as <code>CHECKERR</code> and is located in the <code>util.c</code> file.
Java	Any SQL error is thrown as an <code>SQLException</code> and handled in the catch block of the application.
COBOL	<code>CHECKERR</code> is an external program named <code>checkerr.cbl</code> .
FORTTRAN	<code>CHECKERR</code> is a subroutine located in the <code>util.f</code> file.
REXX	<code>CHECKERR</code> is a procedure located at bottom of the current program.

See “Using `GET ERROR MESSAGE` in Example Programs” on page 120 for the source code for this error checking utility.

Example Client Application: Outcli.java

```
import java.sql.*;           // JDBC classes

class Outcli
{
    static
    {
        try
        {
            Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver").newInstance ();
        }
        catch (Exception e)
        {
            System.out.println ("\n Error loading DB2 Driver...\n" + e);
            System.exit(1);
        }
    }

    public static void main (String argv[])
    {
        try
        {
            System.out.println (" Java Output Stored Procedure Sample");
            // Connect to Sample database

            Connection con = null;
            // URL is jdbc:db2:dbname
            String url = "jdbc:db2:sample";

            if (argv.length == 0) 2
            {
                // connect with default id/password
                con = DriverManager.getConnection(url);
            }
            else if (argv.length == 2)
            {
                String userid = argv[0];
                String passwd = argv[1];

                // connect with user-provided username and password
                con = DriverManager.getConnection(url, userid, passwd);
            }
            else
            {
                throw new Exception("\nUsage: java Outcli [username password]\n");
            }

            // Set AutoCommit
            con.setAutoCommit(true);

            // Register and call the Remote Procedure

            String callName = "outsrv";
            String storedProcName = "Outsrv!outputStoredProcedure";
            String mode = "fenced";

            Statement stmt = con.createStatement (); // for DROP/CREATE

            // drop the stored procedure if it exists
            try
            {
                stmt.executeUpdate ("DROP PROCEDURE " + callName);
            }
            catch (Exception e)
            {
                // ignore this error
            }
        }
    }
}
```



```

    }

    // construct a parameter list for the stored procedure and
    // register it in the system catalogs 3
    String parameterList = "(out    medianSalary    double)";

    System.out.println ("\n Registering Java stored procedure "
        + callName + "\n    as " + storedProcName +
        "\n    in " + mode + " mode");

    stmt.executeUpdate ("CREATE PROCEDURE " + callName + parameterList +
        " LANGUAGE JAVA " +
        " PARAMETER STYLE DB2GENERAL " + mode +
        " EXTERNAL NAME '" + storedProcName + "'");
    stmt.close ();

    // prepare the CALL statement
    CallableStatement callableStmt;
    String sql = "Call " + callName + "(?) ";
    callableStmt = con.prepareCall (sql);

    // register the output parameters 4
    callableStmt.registerOutParameter (1, Types.DOUBLE);

    // set all parameters
    callableStmt.setDouble (1, 0.00);

    // call the stored procedure
    con.setAutoCommit(false); // Enable transactions
    try
    { System.out.println("Calling stored procedure: " + callName );
      callableStmt.execute (); 5

      // Commit the transaction
      con.commit();
    }
    catch (Exception e)
    { // Rollback the transaction
      con.rollback();
      throw e;
    }

    // retrieve output parameters
    double medianSalary = callableStmt.getDouble (1);

    // display the information returned from the stored procedure
    System.out.println ("\n Median Salary: " + medianSalary);

    callableStmt.close (); 6
  }
  catch (Exception e)
  { System.out.println (e);
  }
}
}

```

OUT Stored Procedure Description

1. **Declare Signature.** The procedure returns one value: a variable of type `DOUBLE`. Note that you must specify the OUT host variable as an input parameter in the stored procedure signature.
2. **Turn Off AutoCommit.** To ensure that the stored procedure Connection does not automatically issue any `COMMIT` statements, set the `AutoCommit` value of the Connection instance to **false**.
3. **Declare an Iterator Ordered by Salary.** The stored procedure declares an Iterator with the `ORDER BY SALARY` clause.
4. **Determine Total Number of Employees.** The stored procedure uses a simple `SELECT` statement with the `COUNT` function to retrieve the number of employees in the `EMPLOYEE` table.
5. **FETCH Median Salary.** The stored procedure issues successive `FETCH` statements until it assigns the median salary to a variable.
6. **Return Median Salary.** The stored procedure returns the value of the median salary to the client application.
7. **Return to the Client Application.** Java stored procedures do not return values to the client. If the stored procedure is not written in Java, it returns the value `SQLZ_DISCONNECT_PROC` to the client application if completes successfully, indicating that no further calls to the server procedure will be made.

Example Stored Procedure: Outsrv.sqlj

```
import java.sql.*;           // JDBC classes
import COM.ibm.db2.app.*;    // StoredProc and associated classes
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator Outsrv_Cursor1 (double salary) ;

class Outsrv extends StoredProc
{ // (1) stored procedure body
    public void outputStoredProcure (           1
                                     double medianSalary) throws Exception
    {
        try
        { // Get the DefaultContext
          DefaultContext ctx = DefaultContext.getDefaultContext();

          // If the default context is null, set a new one
          if(ctx == null)
          { // get caller's connection to the database;
            // inherited from StoredProc
            Connection con = getConnection ();
            ctx = new DefaultContext(con);
            DefaultContext.setDefaultContext(ctx);
            con.setAutoCommit(false);           2
          }

          // Declare Variables
          Outsrv_Cursor1 cursor1;
          short          numRecords;
          int             counter = 0;
          double          sal = 0.00;

          // Prepare a Statement to Obtain and Order all Salaries
          #sql cursor1 = { SELECT salary FROM STAFF ORDER BY salary }; 3
          // Determine the Total Number of Records
          #sql { SELECT COUNT(*) INTO :numRecords FROM STAFF };        4
          // Fetch Salaries until the Median Salary is Obtained
          while (counter < numRecords/2 + 1)                             5
          {
              cursor1.next();
              counter++;
          }
          sal = cursor1.salary();
          cursor1.close();

          // set value for the output parameter                           6
          set (1, sal);
        }
        catch (Exception e)
        {
            throw e;
        }
    }
}
```

Code Page Considerations

The code page considerations depend on the server.

When a client program (using, for example, code page A) calls a remote stored procedure that accesses a database using a different code page (for example, code page Z), the following events occur:

1. Input character string parameters (whether defined as host variables or in an SQLDA in the client application) are converted from the application code page (A) to the one associated with the database (Z). Conversion does not occur for data defined in the SQLDA as FOR BIT DATA.
2. Once the input parameters are converted, the database manager does not perform any more code page conversions.

Therefore, **you must run the stored procedure using the same code page as the database**, in this example, code page Z. It is a good practice to prep, compile, and bind the server procedure using the same code page as the database.

3. When the stored procedure finishes, the database manager converts the output character string parameters (whether defined as host variables or in an SQLDA in the client application) and the SQLCA character fields from the database code page (Z) back to the application code page (A). Conversion does not occur for data defined in the SQLDA as FOR BIT DATA.

Note: If the parameter of the stored procedure is defined as FOR BIT DATA at the server, conversion does not occur for a CALL statement to DB2 Universal Database for OS/390 or DB2 Universal Database for AS/400, regardless of whether it is explicitly specified in the SQLDA. (Refer to the section on the SQLDA in the *SQL Reference* for details.)

For more information on this topic, see “Conversion Between Different Code Pages” on page 407.

C++ Consideration

When writing a stored procedure in C++, you may want to consider declaring the procedure name as:

```
extern "C" SQL_API_RC SQL_API_FN proc_name( short *parm1, char *parm2)
```

The extern "C" prevents type decoration (or mangling) of the function name by the C++ compiler. Without this declaration, you have to include all the type decorations for the function name when you call the stored procedure.

Graphic Host Variable Considerations

Any stored procedure written in C or C++, that receives or returns graphic data through its parameter input or output should generally be precompiled with the `WCHARTYPE NOCONVERT` option. This is because graphic data passed through these parameters is considered to be in DBCS format, rather than the `wchar_t` process code format. Using `NOCONVERT` means that graphic data manipulated in SQL statements in the stored procedure will also be in DBCS format, matching the format of the parameter data.

With `WCHARTYPE NOCONVERT`, no character code conversion occurs between the graphic host variable and the database manager. The data in a graphic host variable is sent to, and received from, the database manager as unaltered DBCS characters. Note that if you do not use `WCHARTYPE NOCONVERT`, it is still possible for you to manipulate graphic data in `wchar_t` format in a stored procedure; however, you must perform the input and output conversions manually.

`CONVERT` can be used in `FENCED` stored procedures, and it will affect the graphic data in SQL statements within the stored procedure, but not through the stored procedure's interface. `NOT FENCED` stored procedures must be built using the `NOCONVERT` option.

In summary, graphic data passed to or returned from a stored procedure through its input or output parameters is in DBCS format, regardless of how it was precompiled with the `WCHARTYPE` option.

For important information on handling graphic data in C applications, see “Handling Graphic Host Variables” on page 519. For information on EUC code sets and application guidelines, see “Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations” on page 419, and more specifically to “Considerations for Stored Procedures” on page 424.

Multisite Update Consideration

Stored procedures that applications call with `CONNECT TYPE 2` cannot issue a `COMMIT` or `ROLLBACK`, either dynamically or statically.

Building the Stored Procedure

Stored procedure applications have special compile and link requirements. The client procedure must be part of an executable file, while the stored procedure must be placed in a library on the database server.

When testing these applications, it is helpful to execute both procedures locally on the same machine. (This machine must be configured as a client/server.) After the application executes locally without error, move the stored procedure to a database server, or move the client procedure to a database client for further testing.

Client Application

The precompile, compile, and link requirements of the client application are identical to those of a normal database application.

Refer to the *Application Building Guide* for details on how to build and execute a client application in the programming language you want to use.

Stored Procedure

To build a stored procedure, you must perform the following steps:

1. Precompile the stored procedure source file, with the exception of JDBC or DB2 CLI stored procedures
2. Compile and link the stored procedures to produce a library. The compile and link commands usually require special options.
3. Bind the stored procedure to the database on the server if it contains embedded SQL. You do not have to bind JDBC and DB2 CLI stored procedures.

For all of these steps, the stored procedure must use the same code page as the database.

Refer to the *Application Building Guide* for detailed instructions on building your stored procedures.

NOT FENCED Stored Procedures

Your stored procedure can run as either a *FENCED* or a *NOT FENCED* stored procedure, depending on whether you register the stored procedure as *FENCED* or *NOT FENCED* in the *CREATE PROCEDURE* statement.

A *NOT FENCED* stored procedure runs in the same address space as the database manager (the DB2 Agent's address space). Running your stored procedure as *NOT FENCED* results in increased performance when compared with running it as *FENCED* because *FENCED* stored procedures, by default, run in a special DB2 process. The address space of this process is distinct from the DB2 System Controller.

Notes:

1. While you can expect performance improvements from running NOT FENCED stored procedures, user code can accidentally or maliciously damage the database control structures. You should only use NOT FENCED stored procedures when you need to maximize the performance benefits. Test all your stored procedures thoroughly prior to running them as NOT FENCED.
2. If a severe error does occur while you are running a NOT FENCED stored procedure, the database manager determines whether the error occurred in the stored procedure code or the database code, and attempts an appropriate recovery.

For debugging purposes, consider using *local FENCED stored procedures*. NOT FENCED and regular FENCED stored procedures complicate your debugging efforts by giving your debugger access to additional address space. Local FENCED stored procedures run in the application's address space and allow your debugger to access both the application code and the stored procedure code. To enable local FENCED stored procedures for debugging, perform the following steps:

1. Register the stored procedure as a FENCED stored procedure.
2. Set the DB2_STPROC_ALLOW_LOCAL_FENCED registry variable to true. For information on registry variables, refer to the *Administration Guide, Performance*.
3. Run the client application on the same machine as the DB2 server.

Note: While debugging a local FENCED stored procedure, be careful not to introduce statements that violate the restrictions listed in “Restrictions” on page 198. DB2 treats calls to local FENCED stored procedures as calls to a subroutine of the client application. Therefore, local FENCED stored procedures can contain statements that violate restrictions on normal stored procedures, like performing CONNECT or COMMIT statements in the procedure body.

When you are writing a NOT FENCED stored procedure, keep in mind that it may run in a threaded environment, depending on the operating system. Thus, the stored procedure must either be completely re-entrant, or manage its static variables so that access to these variables is serialized.

NOT FENCED stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. See “The WCHARTYPE Precompiler Option” on page 521 for more information.

The following NOT FENCED stored procedures are **not** supported by DB2:

- 16-bit

- REXX
- Multi-threaded

The following DB2 APIs and any DB2 CLI API are **not** supported in a NOT FENCED stored procedure:

- BIND
- EXPORT
- IMPORT
- PRECOMPILE PROGRAM
- ROLLFORWARD DATABASE

Returning Result Sets from Stored Procedures

You can code stored procedures to return one or more result sets to DB2 CLI, ODBC, JDBC, or SQLJ client applications. Aspects of this support include:

- Only DB2 CLI, ODBC, JDBC, and SQLJ clients can accept multiple result sets.
- DB2 clients that use embedded SQL can accept multiple result sets if the stored procedure resides on a server that is accessible from a DataJoiner Version 2 server. Stored procedures on host and AS/400 platforms can return multiple result sets to DB2 Connect clients. Stored procedures on DB2 Universal Database servers can return multiple result sets to host and AS/400 clients. Consult the product documentation for DataJoiner or the host or AS/400 platform for more information.
- The client application program can describe the result sets returned.
- Result sets must be processed in serial fashion by the application. A cursor is automatically opened on the first result set and a special call (SQLMoreResults for DB2 CLI, getMoreResults for JDBC, getNextResultSet for SQLJ) is provided to both close the cursor on one result set and to open it on the next.
- The stored procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on that result set, and leaving the cursor open when exiting the procedure. If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened
- Only unread or unfetched rows are passed back in the result set.
- Stored procedures which return result sets must be run in FENCED mode.
- A COMMIT or ROLLBACK will close all cursors except WITH HOLD cursors.

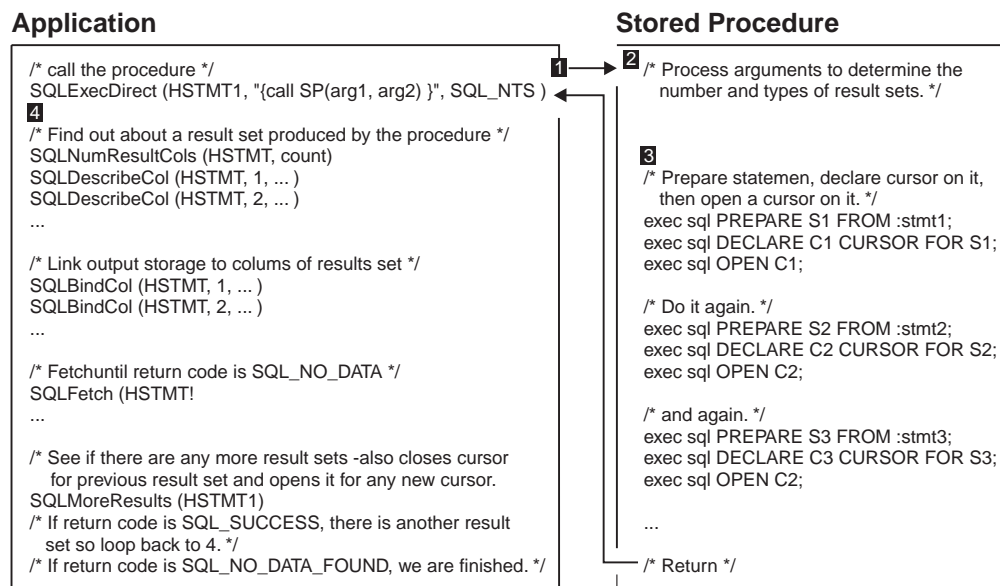
- The `RESULT_SETS` column in the `DB2CLI.PROCEDURES` table indicates whether or not a stored procedure can return result sets. Only procedures with a value greater than zero in this column will return result sets for open cursors.

For additional details on handling multiple result sets:

- in DB2 CLI, refer to the *CLI Guide and Reference*
- in Java, refer to the DB2 Java Enablement web page at <http://www.software.ibm.com/data/db2/java/> for links to the JDBC and SQLJ specifications

The following diagram shows typical use of this facility:

- 1** A client application program AP calls a procedure SP providing two input arguments.
- 2** SP determines, based on those arguments, that three result sets are to be returned.
- 3** SP provides open cursors on the statements that define the result sets.
- 4** AP processes the result sets in a general loop that is able to deal with different numbers and different types of results.



Resolving Problems

If a stored procedure application fails to execute properly, ensure that:

- The stored procedure is built using the correct calling sequence, compile options, and so on.
- The application executes locally with both client application and stored procedure on the same workstation.
- The stored procedure is stored in the proper location in accordance with the instructions in the *Application Building Guide*.

For example, in an OS/2 environment, the dynamic link library for a FENCED stored procedure is located in the *instance_name*\function directory on the database server.

- The application, except if it is written in DB2 CLI and JDBC, is bound to the database.
- The stored procedure accurately returns any SQLCA error information to the client application.
- Stored procedure function names are *case-sensitive*, and must match exactly on client and server.
- If you register the stored procedure with a CREATE PROCEDURE statement, stored procedure function names must not match their library name.

For example, the database manager will execute the stored procedure myfunc contained in the Windows 32-bit operating system library myfunc.dll as a DB2DARI function, disregarding the values specified in its associated CREATE PROCEDURE statement.

Note: For more information on debugging Java stored procedures, see “Debugging Stored Procedures” on page 559.

You can use the debugger supplied with your compiler to debug a local FENCED stored procedure as you would any other application. Consult your compiler documentation for information on using the supplied debugger.

For example, to use the debugger supplied with Visual Studio™ on Windows NT, perform the following steps:

- Step 1. Set the DB2_STPROC_ALLOW_LOCAL_FENCED registry variable to true.
- Step 2. Compile the source file for the stored procedure DLL with the -Zi and -Od flags, and then link the DLL using the -DEBUG option.
- Step 3. Copy the resulting DLL to the *instance_name* \function directory of the server.

- Step 4. Invoke the client application on the server with the Visual Studio debugger. For the client application `outcli.exe`, enter the following command:
- ```
msdev outcli.exe
```
- Step 5. When the Visual Studio debugger window opens, select **Project -> Settings**.
- Step 6. Click the **Debug** tab.
- Step 7. Click the **Category** arrow and select the **Additional DLLs**.
- Step 8. Click the **New** button to create a new module.
- Step 9. Click the **Browse** button to open the **Browse** window.
- Step 10. Select the module `outsrv.dll` and click **OK** to close the **Settings** window.
- Step 11. Open the source file for the stored procedure and set a breakpoint.
- Step 12. Click the **Go** button. The Visual Studio debugger stops when the stored procedure is invoked.
- Step 13. At this point, you can debug the stored procedure using the Visual Studio debugger.

Refer to the Visual Studio product documentation for further information on using the Visual Studio debugger.

---

## Overview of Stored Procedure Builder

To use a stored procedure, you write two programs: the stored procedure itself, which runs on a database server, and a client application, which runs on a client workstation. The client application calls the stored procedure by using one of the available API methods.

IBM DB2 Stored Procedure Builder (SPB) assists you with creating a stored procedure that runs on a database server. You must write the client application separately.

### What is SPB?

SPB is a graphical application that supports the rapid development of DB2 stored procedures. Using SPB, you can perform the following tasks:

- Create new stored procedures
- Build stored procedures on local and remote DB2 servers
- Modify and rebuild existing stored procedures
- Test and debug the execution of installed stored procedures

SPB provides a single development environment that supports the entire DB2 family ranging from the workstation to System/390. You can launch SPB as a separate application from the DB2 Universal Database program group, or you can launch SPB from any of the following development applications:

- Microsoft Visual Studio
- Microsoft Visual Basic
- IBM VisualAge for Java

SPB is implemented with Java and all database connections are managed by using Java Database Connectivity (JDBC). Using a JDBC driver, you can connect to any local DB2 alias or any other database for which you can specify a host, port, and database name. Several JDBC drivers are installed with SPB.

**Note:** To use SPB, you must be connected to a DB2 database for development.

For more information about using SPB, refer to the IBM DB2 Stored Procedure Builder Help.

## **Advantages of Using SPB**

SPB provides an easy-to-use development environment for creating, installing, and testing stored procedures, so that you can focus on creating your stored procedure logic rather than the details of registering, building, and installing stored procedures on a DB2 server. Additionally, with SPB, you can develop stored procedures on one operating system and build them on other server operating systems.

### **Creating New Stored Procedures**

Using SPB greatly simplifies the process of creating and installing stored procedures on a DB2 database server. The new procedure SmartGuide and SQL Assistant facilitate the development of stored procedures.

In SPB, you create stored procedures in Java. As a result, the stored procedures are highly portable among operating systems. Using the new procedure SmartGuide, you create your basic SQL structure; then, you can use your editor to modify the stored procedure so that it contains highly sophisticated stored procedure logic.

When creating a stored procedure, you can choose to return a single result set, multiple result sets, or output parameters only. You might choose not to return a result set when your stored procedure creates or updates database tables. You can use the new procedure SmartGuide to define input and output parameters for a stored procedure so that it receives values for host variables

from the client application. Additionally, you can create multiple SQL statements in a stored procedure; the stored procedure receives a case value and selects one of a number of queries.

Building a stored procedure on a target database is as simple as clicking Finish in the new procedure SmartGuide. You no longer have to manually register the stored procedure with DB2 by using the CREATE PROCEDURE statement.

### **Working with Existing Stored Procedures**

After you successfully build a stored procedure on a database server, you are ready to modify, rebuild, run, and test the procedure. By modifying a stored procedure, you can add methods to the code to include sophisticated stored procedure logic. When you open a stored procedure, the stored procedure source code is displayed in the details pane on the right side. The details pane has a language sensitive editor for stored procedures written by using Java.

Running a stored procedure from within SPB allows you to test the procedure to make sure that it is correctly installed. When you run a stored procedure, depending on how you set it up, it can return result sets based on test input parameter values that you enter. Testing stored procedures makes programming the client application easier because you know that the stored procedure is correctly installed on the DB2 database server; you can focus on writing and debugging the client application.

From the Project window, you can easily drop a stored procedure or copy it to another database connection.

### **Creating SPB Projects**

When you open a new or existing SPB project, the Project window shows all the stored procedures that reside on the DB2 database to which you are connected. You can choose to filter stored procedures so that you view procedures based on their name or schema. An SPB project saves only connection information and stored procedure objects that have not been successfully built.

### **Debugging Stored Procedures**

Using SPB and the debug client, you can remotely debug a stored procedure installed on a DB2 server. To debug a stored procedure, you run the stored procedure in debug mode. You are not required to debug the stored procedures from within an application program. You can separate testing your stored procedure from testing the calling application program.

Using the Debug Settings notebook in SPB, you can view all the stored procedures that you have the authority to change, add, or remove debug entries in the stored procedures debug table. If you are a DBA or you created the selected stored procedure, you can grant authorization to other users to debug the stored procedure.

---

## Part 4. Object-Relational Programming





---

# Chapter 7. Using the Object-Relational Capabilities

|                                          |     |                                     |     |
|------------------------------------------|-----|-------------------------------------|-----|
| Why Use the DB2 Object Extensions? . . . | 221 | Example: Exploiting LOB Function to |     |
| DB2 Approach to Supporting Objects . . . | 222 | Populate the Database . . . . .     | 225 |
| Combining UDTs, UDFs, and LOBs . . .     | 223 | Example: Exploiting UDFs to Query   |     |
| Examples of Complex Applications. . . .  | 223 | Instances of UDTs . . . . .         | 225 |
| Example: Defining the UDT and UDFs       | 223 |                                     |     |

---

## Why Use the DB2 Object Extensions?

One of the most important recent developments in modern programming language technology is *object-orientation*. Object-orientation is the notion that entities in the application domain can be modeled as independent objects that are related to one another by means of classification. Functions (methods) encapsulate the external behavior and characteristics of objects by hiding their internal implementation details. Object-orientation lets you capture the similarities and differences among objects in your application domain and group those objects together into related types. Objects of the same type behave in the same way because they share the same set of type-specific functions, reflecting the behavior of your objects in the application domain.

With the object extensions of DB2, you can incorporate object-oriented (OO) concepts and methodologies into your relational database by extending it with richer sets of types and functions. With these extensions, you can store instances of object-oriented data types in columns of tables, and operate on them by means of functions in SQL statements. In addition, you can make the semantic behavior of stored objects an important resource that can be shared among all your applications by means of the database.

To incorporate object-orientation into your relational database systems, you can define new types and functions of your own to reflect the semantics of the objects in your application domain. Some of the data objects you want to model may be large and complex (for example, text, voice, image, and financial data). Therefore, you may also need mechanisms for the storage and manipulation of large objects. User-defined Distinct types (UDTs), user-defined functions (UDFs), and Large Objects (LOBs) are the mechanisms provided by DB2 to fulfill these needs. With DB2, you can now define new types and functions of your own to store and manipulate application objects, potentially gigabytes in size, within the database.

UDTs, UDFs, and LOBs build the foundation for the DB2 object-relational extensions. In addition to UDTs, UDFs, and LOBs, DB2 also includes mechanisms that enable the representation of another important piece of the object semantics: the *rules* that govern the kinds of operations that are allowed

on such objects in the application domain. These rules are embodied in the DB2 triggers and constraints mechanisms.

As described in subsequent sections, there is an important synergy among these object-oriented features. You can model a complex object in the application domain as a UDT. The UDT may in turn be internally represented as a LOB. The UDT's behavior may be implemented in terms of UDFs, and its integrity rules implemented in terms of constraints and triggers. This section shows you the steps required to define UDTs and UDFs, how to use LOBs, and how UDTs, UDFs, and LOBs can better represent the objects in your application and thus work together. "Chapter 12. Using Triggers in an Active DBMS" on page 373 shows you how triggers can enhance the semantics of your applications.

The use of the DB2 object-oriented mechanisms (UDTs, UDFs, LOBs, and triggers) is not restricted to the support of object-oriented applications. Just as C++, a popular object-oriented programming language, is used to implement all sorts of non-object-oriented applications, the object-oriented mechanisms provided by DB2 are also very useful to support all kinds of non-object-oriented applications. UDTs, UDFs, LOBs, triggers, and constraints are general-purpose mechanisms that can be used to model any database application. For this reason, these DB2 object extensions offer extensive support for both non-traditional, that is, object-oriented applications, in addition to improving support for traditional ones.

---

## DB2 Approach to Supporting Objects

The object extensions of DB2 enable you to realize many of the benefits of object technology while building on the strengths of relational technology. In a relational system, data types are used to describe the data stored in columns of tables where the instances (or objects) of these data types are stored. Operations on these instances are supported by means of operators or functions that can be invoked anywhere that expressions are allowed.

The DB2 approach to support object extensions fits exactly into the relational paradigm. UDTs are data types defined by the user which, like built-in types, can be used to describe the data stored in columns of tables. UDFs are functions defined by a user which, like built-in functions or operators, support the manipulation of UDT instances. Thus, UDT instances are stored in columns of tables and manipulated by UDFs in SQL queries. UDTs can be internally represented in different ways. LOBs are just one example of this.

---

## Combining UDTs, UDFs, and LOBs

According to the concept of object-orientation, similar objects in the application domain are grouped into related types. Each of these types have a name, an internal representation, and behavior.

- By using UDTs, you can tell DB2 the name of your new type and how it is internally represented.
- A LOB is one of the possible internal representations for your new type and is the most suitable representation for large, complex structures.
- By using UDFs, you can define the behavior of the new type.

Consequently, there is an important synergy between UDTs, UDFs, and LOBs. An application type with a complex data structure and behavior is modeled as a UDT that is internally represented as a LOB, with its behavior implemented by UDFs. As described in “Chapter 12. Using Triggers in an Active DBMS” on page 373, the rules governing the semantic integrity of your application type will be represented as constraints and triggers. To have better control and organization of your related UDTs and UDFs, you should keep them in the same schema.

---

## Examples of Complex Applications

The following examples show how you can use UDTs, UDFs, and LOBs together in complex applications:

- Example: Defining the UDT and UDFs
- Example: Exploiting LOB Function to Populate the Database
- Example: Exploiting UDFs to Query Instances of UDTs

The following examples use type BLOB (Binary Large Object) data with a length of one megabyte as the base type of a UDT. The examples also use type BLOB host variables. BLOBs and the other types of LOB, as well as a technique for reducing LOB byte transfer between DB2 and the application, are covered in detail in “Chapter 8. Using Large Objects (LOBs)” on page 227.

### Example: Defining the UDT and UDFs

Suppose you would like to keep the electronic mail (e-mail) sent to your company in DB2 tables. Ignoring any issues of privacy, you are planning to write queries over such e-mail to find out their subject, how often your e-mail service is being used to receive customer orders, and so on.

Because e-mail can be quite large, and because it has a complex internal structure (a sender, a receiver, the subject, date, and the e-mail content), you decide to represent the e-mail by means of a UDT whose source type is a large object.

You define a set of UDFs on your e-mail type, such as functions to extract the subject of the e-mail, the sender, the date, and so on. You also define functions that can perform searches on the content of the e-mail.

To create these UDFs, you must:

1. Write the UDFs in a supported language (see “Chapter 11. Writing User-Defined Functions (UDFs)” on page 287 for more information)
2. Define the UDFs using the following CREATE statements:

```
CREATE DISTINCT TYPE E_MAIL AS BLOB (1M)

CREATE FUNCTION SUBJECT (E_MAIL)
 RETURNS VARCHAR (200)
 EXTERNAL NAME '/u/mail/fncdir/e_mail!subject'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION

CREATE FUNCTION SENDER (E_MAIL)
 RETURNS VARCHAR (200)
 EXTERNAL NAME '/u/mail/fncdir/e_mail!sender'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION

CREATE FUNCTION RECEIVER (E_MAIL)
 RETURNS VARCHAR (200)
 EXTERNAL NAME '/u/mail/fncdir/e_mail!receiver'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION

CREATE FUNCTION SENDING_DATE (E_MAIL)
 RETURNS DATE CAST FROM VARCHAR(10)
 EXTERNAL NAME '/u/mail/fncdir/e_mail!sending_date'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION
```

```

CREATE FUNCTION CONTENTS (E_MAIL)
 RETURNS BLOB (1M)
 EXTERNAL NAME '/u/mail/funcdir/e_mail!contents'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION

CREATE FUNCTION CONTAINS (E_MAIL, VARCHAR (200))
 RETURNS INTEGER
 EXTERNAL NAME '/u/mail/funcdir/e_mail!contains'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION

CREATE TABLE ELECTRONIC_MAIL
 (ARRIVAL_TIMESTAMP TIMESTAMP,
 MESSAGE E_MAIL)

```

### Example: Exploiting LOB Function to Populate the Database

Suppose you populate your table by transferring your e-mail that is maintained in files into DB2. You would execute the following INSERT statement multiple times with different values of the HV\_EMAIL\_FILE until you have stored all your e-mail into DB2:

```

EXEC SQL BEGIN DECLARE SECTION
 SQL TYPE IS BLOB_FILE HV_EMAIL_FILE;

EXEC SQL END DECLARE SECTION
 strcpy (HV_EMAIL_FILE.NAME, "/u/mail/email/mbox");
 HV_EMAIL_FILE.NAME_LENGTH = strlen(HV_EMAIL_FILE.NAME);
 HV_EMAIL_FILE.FILE_OPTIONS = 2;

EXEC SQL INSERT INTO ELECTRONIC_MAIL
 VALUES (CURRENT TIMESTAMP, :hv_email_file);

```

Because all the function provided by DB2 LOB support is applicable to UDTs whose source type are LOBs, you have used LOB file reference variables to assign the contents of the file into the UDT column. You have not used the cast function to convert values of BLOB type into your e-mail type because DB2 let you assign values of the source type of a distinct type to targets to the distinct type.

### Example: Exploiting UDFs to Query Instances of UDTs

Suppose you need to know how much e-mail was sent by a specific customer regarding customer orders and you have the e-mail address of your customers in the customers table.

```

SELECT COUNT (*)
 FROM ELECTRONIC_MAIL AS EMAIL, CUSTOMERS
 WHERE SUBJECT (EMAIL.MESSAGE) = 'customer order'
 AND CUSTOMERS.EMAIL_ADDRESS = SENDER (EMAIL.MESSAGE)
 AND CUSTOMERS.NAME = 'Customer X'

```

You have used the UDFs defined on the UDT in this SQL query since they are the only means to manipulate the UDT. In this sense, your UDT e-mail is completely encapsulated. That is, its internal representation and structure are hidden and can only be manipulated by the defined UDFs. These UDFs know how to interpret the data without the need to expose its representation.

Suppose you need to know the details of all the e-mail your company received in 1994 which had to do with the performance of your products in the marketplace.

```

SELECT SENDER (MESSAGE), SENDING_DATE (MESSAGE), SUBJECT (MESSAGE)
 FROM ELECTRONIC_MAIL

 WHERE CONTAINS (MESSAGE,

 '"performance" AND "products" AND "marketplace"') = 1

```

You have used the contains UDF which is capable of analyzing the contents of the message searching for relevant keywords or synonyms.

---

## Chapter 8. Using Large Objects (LOBs)

|                                                                      |     |                                                      |     |
|----------------------------------------------------------------------|-----|------------------------------------------------------|-----|
| What are LOBs? . . . . .                                             | 227 | How the Sample LOBEVAL Program Works . . . . .       | 240 |
| Understanding Large Object Data Types (BLOB, CLOB, DBCLOB) . . . . . | 228 | C Sample: LOBEVAL.SQC. . . . .                       | 241 |
| Understanding Large Object Locators . . . . .                        | 229 | COBOL Sample: LOBEVAL.SQB. . . . .                   | 243 |
| Example: Using a Locator to Work With a CLOB Value . . . . .         | 231 | Indicator Variables and LOB Locators . . . . .       | 246 |
| How the Sample LOBLOC Program Works . . . . .                        | 231 | LOB File Reference Variables. . . . .                | 246 |
| C Sample: LOBLOC.SQC . . . . .                                       | 232 | Example: Extracting a Document To a File . . . . .   | 248 |
| COBOL Sample: LOBLOC.SQB . . . . .                                   | 234 | How the Sample LOBFILE Program Works . . . . .       | 248 |
| FORTRAN Sample: LOBLOC.SQF . . . . .                                 | 237 | C Sample: LOBFILE.SQC . . . . .                      | 249 |
| Example: Deferring the Evaluation of a LOB Expression . . . . .      | 239 | COBOL Sample: LOBFILE.SQB . . . . .                  | 251 |
|                                                                      |     | Example: Inserting Data Into a CLOB Column . . . . . | 253 |

---

### What are LOBs?

The LONG VARCHAR and LONG VARGRAPHIC data types have a limit of 32K bytes of storage. While this may be sufficient for small to medium size text data, applications often need to store large text documents. They may also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images. DB2 provides three data types to store these data objects as strings of up to two (2) gigabytes (GB) in size. The three data types are: Binary Large Objects (BLOBs), single-byte Character Large Objects (CLOBs), and Double-Byte Character Large Objects (DBCLOBs).

Along with storing large objects (LOBs), a way is also needed to refer to, and to use and modify, each LOB in the database. Each DB2 table may have a large amount of associated LOB data. Although any single LOB value may not exceed 2 gigabytes, a single row may contain as much as 24 gigabytes of LOB data, and a table may contain as much as 4 terabytes of LOB data. The content of the LOB column of a particular row at any point in time has a *large object value*.

You can refer to and manipulate LOBs using host variables just as you would any other data type. However, host variables use the client memory buffer which may not be large enough to hold LOB values. Other means are necessary to manipulate these large values. *Locators* are useful to identify and manipulate a large object value at the database server and for extracting pieces of the LOB value. *File reference variables* are useful for physically moving a large object value (or a large part of it) to and from the client.

**Note:** DB2 offers LOB support for JDBC and SQLJ applications. For more information on using LOBs in Java applications, see “JDBC 2.0” on page 544.

The subsections that follow discuss in more detail those topics introduced above.

---

## Understanding Large Object Data Types (BLOB, CLOB, DBCLOB)

Large object data types store data ranging in size from zero bytes to two gigabytes - 1.

The three large object data types have the following definitions:

- **Character Large Objects (CLOBs)** — A character string made up of single-byte characters with an associated code page. This data type is best for holding text-oriented information where the amount of information could grow beyond the limits of a regular VARCHAR data type (upper limit of 4K bytes). Code page conversion of the information is supported as well as compatibility with the other character types.
- **Double-Byte Character Large Objects (DBCLOBs)** — A character string made up of double-byte characters with an associated code page. This data type is best for holding text-oriented information where double-byte character sets are used. Again, code page conversion of the information is supported as well as compatibility with the other character types.
- **Binary Large Objects (BLOBs)** — A binary string made up of bytes with no associated code page. This data type may be the most useful because it can store binary data, making it a perfect source type for use by User-defined Distinct Types (UDTs). UDTs using BLOBs as the source type are created to store image, voice, graphical, and other types of business or application specific data. For more information on UDTs, see “Chapter 10. User-defined Distinct Types (UDTs)” on page 275.

A separate database location stores all large object values outside their records in the table. There is a *large object descriptor* for each large object in each row in a table. The large object descriptor contains control information used to access the large object data stored elsewhere on disk. It is the storing of large object data outside their records that allows LOBs to be 2 GB in size. Accessing the large object descriptor causes a small amount of overhead when manipulating LOBs. (For storage and performance reasons you would likely not want to put small data items into LOBs.)

The maximum size for each large object column is part of the declaration of the large object type in the CREATE TABLE statement. The maximum size of a large object column determines the maximum size of any LOB descriptor in



that column. As a result, it also determines how many columns of all data types can fit in a single row. The space used by the LOB descriptor in the row ranges from approximately 60 to 300 bytes, depending on the maximum size of the corresponding column. For specific sizes of the LOB descriptor, refer to the CREATE TABLE statement in the *SQL Reference*.

The lob-options-clause on CREATE TABLE gives you the choice to log (or not) the changes made to the LOB column(s). This clause also allows for a compact representation for the LOB descriptor (or not). This means you can allocate only enough space to store the LOB or you can allocate extra space for future append operations to the LOB. The tablespace-options-clause allows you to identify a LONG table space to store the column values of long field or LOB data types. For more information on the CREATE TABLE and ALTER TABLE statements, refer to the *SQL Reference*.

With their potentially very large size, LOBs can slow down the performance of your database system significantly when moved into or out of a database. Even though DB2 does not allow logging of a LOB value greater than 1 GB, LOB values with sizes near several hundred megabytes can quickly push the database log to near capacity. An error, SQLCODE -355 (SQLSTATE 42993), results from attempting to log a LOB greater than 1 GB in size. The lob-options-clause in the CREATE TABLE and ALTER TABLE statements allows users to turn off logging for a particular LOB column. Although setting the option to NOT LOGGED improves performance, changes to the LOB values after the most recent backup are lost during roll-forward recovery. For more information on these topics, refer to the *Administration Guide*.

---

## Understanding Large Object Locators

Conceptually, LOB locators represent a simple idea that has been around for a while; use a small, easily managed value to refer to a much larger value. Specifically, a LOB locator is a 4-byte value stored in a host variable that a program can use to refer to a LOB value (or LOB expression) held in the database system. Using a LOB locator, a program can manipulate the LOB value as if the LOB value was stored in a regular host variable. The difference in using the LOB locator is that there is no need to transport the LOB value from the server to the application (and possibly back again).

The LOB locator is associated with a LOB value or LOB expression, not a row or physical storage location in the database. Therefore, after selecting a LOB value into a locator, there is no operation that you could perform on the original row(s) or tables(s) that would have any effect on the value referenced by the locator. The value associated with the locator is valid until the unit of work ends, or the locator is explicitly freed, whichever comes first. The FREE

LOCATOR statement releases a locator from its associated value. In a similar way, a commit or roll-back operation frees all LOB locators associated with the transaction.

LOB locators can also be passed between DB2 and UDFs. There are special APIs available for UDFs to manipulate the LOB values using LOB locators. For more information on these APIs see “Using LOB Locators as UDF Parameters or Results” on page 335.

When selecting a LOB value, you have three options:

- Select the entire LOB value into a host variable. The entire LOB value is copied from the server to the client.
- Select just a LOB locator into a host variable. The LOB value remains on the server; the LOB locator moves to the client.
- Select the entire LOB value into a file reference variable. The LOB value is moved to a file at the client without going through the application’s memory.

The use of the LOB value within the program can help the programmer determine which method is best. If the LOB value is very large and is needed only as an input value for one or more subsequent SQL statements, then it is best to keep the value in a locator. The use of a locator eliminates any client/server communication traffic needed to transfer the LOB value to the host variable and back to the server.

If the program needs the entire LOB value regardless of the size, then there is no choice but to transfer the LOB. Even in this case, there are still three options available to you. You can select the entire value into a regular or file host variable, but it may also work out better to select the LOB value into a locator and read it piecemeal from the locator into a regular host variable, as suggested in the following example.

---

## Example: Using a Locator to Work With a CLOB Value

In this example, the application program retrieves a locator for a LOB value; then it uses the locator to extract the data from the LOB value. Using this method, the program allocates only enough storage for one piece of LOB data (the size is determined by the program) and it needs to issue only one fetch call using the cursor.

### How the Sample LOBLOC Program Works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. CLOB LOCATOR host variables are declared.
2. **Fetch the LOB value into the host variable LOCATOR.** A CURSOR and FETCH routine is used to obtain the location of a LOB field in the database to a host variable locator.
3. **Free the LOB LOCATORS.** The LOB LOCATORS used in this example are freed, releasing the locators from their previously associated values.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

**C**                    check\_error is redefined as CHECKERR and is located in the util.c file.

**COBOL**            CHECKERR is an external program named checkerr.cbl.

**FORTRAN**        CHECKERR is a subroutine located in the util.f file.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

## C Sample: LOBLOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

 EXEC SQL BEGIN DECLARE SECTION; 1
 char number[7];
 long deptInfoBeginLoc;
 long deptInfoEndLoc;
 SQL TYPE IS CLOB_LOCATOR resume;
 SQL TYPE IS CLOB_LOCATOR deptBuffer;
 short lobind;
 char buffer[1000]="";
 char userid[9];
 char passwd[19];
 EXEC SQL END DECLARE SECTION;

#ifdef DB268K
 /* Before making any API calls for 68K environment,
 need to initial the Library Manager */
 InitLibraryManager(0,kCurrentZone,kNormalMemory);
 atexit(CleanupLibraryManager);
#endif

 printf("Sample C program: LOBLOC\n");

 if (argc == 1) {
 EXEC SQL CONNECT TO sample;
 CHECKERR ("CONNECT TO SAMPLE");
 }
 else if (argc == 3) {
 strcpy (userid, argv[1]);
 strcpy (passwd, argv[2]);
 EXEC SQL CONNECT TO sample USER :userid USING :passwd;
 CHECKERR ("CONNECT TO SAMPLE");
 }
 else {
 printf ("USAGE: lobloc [userid passwd]");
 return 1;
 } /* endif */

 /* Employee A10030 is not included in the following select, because
 the lobeval program manipulates the record for A10030 so that it is
```

```

not compatible with lobloc */

EXEC SQL DECLARE c1 CURSOR FOR
 SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
 AND empno <> 'A00130';

EXEC SQL OPEN c1;
CHECKERR ("OPEN CURSOR");

do {
 EXEC SQL FETCH c1 INTO :number, :resume :lobind; 2
 if (SQLCODE != 0) break;
 if (lobind < 0) {
 printf ("NULL LOB indicated\n");
 } else {
 /* EVALUATE the LOB LOCATOR */
 /* Locate the beginning of "Department Information" section */
 EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
 INTO :deptInfoBeginLoc;
 CHECKERR ("VALUES1");

 /* Locate the beginning of "Education"
 section (end of "Dept.Info") */
 EXEC SQL VALUES (POSSTR(:resume, 'Education'))
 INTO :deptInfoEndLoc;
 CHECKERR ("VALUES2");

 /* Obtain ONLY the "Department Information"
 section by using SUBSTR */
 EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
 :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
 CHECKERR ("VALUES3");

 /* Append the "Department Information" section
 to the :buffer var. */
 EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
 CHECKERR ("VALUES4");
 } /* endif */
} while (1);

printf ("%s\n",buffer);

EXEC SQL FREE LOCATOR :resume, :deptBuffer; 3
CHECKERR ("FREE LOCATOR");

EXEC SQL CLOSE c1;
CHECKERR ("CLOSE CURSOR");

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : LOBLOC.SQC */

```

## COBOL Sample: LOBLOC.SQB

Identification Division.  
Program-ID. "lobloc".

Data Division.  
Working-Storage Section.

copy "sqlenv.cbl".  
copy "sql.cbl".  
copy "sqlca.cbl".

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 userid pic x(8).
01 passwd.
 49 passwd-length pic s9(4) comp-5 value 0.
 49 passwd-name pic x(18).
01 empnum pic x(6).
01 di-begin-loc pic s9(9) comp-5.
01 di-end-loc pic s9(9) comp-5.
01 resume USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 di-buffer USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 lobind pic s9(4) comp-5.
01 buffer USAGE IS SQL TYPE IS CLOB(1K).
EXEC SQL END DECLARE SECTION END-EXEC.
```

**1**

77 errloc pic x(80).

Procedure Division.

Main Section.

display "Sample COBOL program: LOBLOC".

\* Get database connection information.

display "Enter your user id (default none): "  
with no advancing.  
accept userid.

if userid = spaces

EXEC SQL CONNECT TO sample END-EXEC

else

display "Enter your password : " with no advancing  
accept passwd-name.

\* Passwords in a CONNECT statement must be entered in a VARCHAR

\* format with the length of the input string.

inspect passwd-name tallying passwd-length for characters  
before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd  
END-EXEC.

move "CONNECT TO" to errloc.

call "checkerr" using SQLCA errloc.

\* Employee A10030 is not included in the following select, because

\* the lobeval program manipulates the record for A10030 so that it

\* it is not compatible with lobloc

```

EXEC SQL DECLARE c1 CURSOR FOR
 SELECT empno, resume FROM emp_resume
 WHERE resume_format = 'ascii'
 AND empno <> 'A00130' END-EXEC.

EXEC SQL OPEN c1 END-EXEC.
move "OPEN CURSOR" to errloc.
call "checkerr" using SQLCA errloc.

Move 0 to buffer-length.

perform Fetch-Loop thru End-Fetch-Loop
 until SQLCODE not equal 0.

* display contents of the buffer.
 display buffer-data(1:buffer-length).

EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC.
move "FREE LOCATOR" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL CLOSE c1 END-EXEC.
move "CLOSE CURSOR" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Main.
 go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :empnum, :resume :lobind
END-EXEC.

 if SQLCODE not equal 0
 go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
 if lobind less than 0 go to NULL-lob-indicated.

* Value exists. Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
 EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
 INTO :di-begin-loc END-EXEC.
 move "VALUES1" to errloc.
 call "checkerr" using SQLCA errloc.

* Locate the beginning of "Education" section (end of Dept.Info)
 EXEC SQL VALUES (POSSTR(:resume, 'Education'))
 INTO :di-end-loc END-EXEC.
 move "VALUES2" to errloc.
 call "checkerr" using SQLCA errloc.

```

**3**

**2**

```

 subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
 EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
 :di-end-loc))
 INTO :di-buffer END-EXEC.
 move "VALUES3" to errloc.
 call "checkerr" using SQLCA errloc.

* Append the "Department Information" section to the :buffer var
 EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
 END-EXEC.
 move "VALUES4" to errloc.
 call "checkerr" using SQLCA errloc.

 go to End-Fetch-Loop.

NULL-lob-indicated.
 display "NULL LOB indicated".

End-Fetch-Loop. exit.

End-Prog.
 stop run.

```



## FORTRAN Sample: LOBLOC.SQF

```

 program lobloc
 implicit none

 include 'sqlenv.f'
 include 'sql.f'
 EXEC SQL INCLUDE SQLCA

 EXEC SQL BEGIN DECLARE SECTION

* 1
 character*8 userid
 character*18 passwd
 character*6 empnum
 integer*4 di_begin_loc
 integer*4 di_end_loc
 SQL TYPE IS CLOB_LOCATOR resume
 SQL TYPE IS CLOB_LOCATOR di_buffer
 integer*2 lobind
 SQL TYPE IS CLOB(1K) buffer
 EXEC SQL END DECLARE SECTION

 character*80 errloc

 print *, 'Sample Fortran Program: LOBLOC'

 print *, 'Enter your user id (default none):'
 read 101, userid
101 format (a8)

 if(userid(1:1) .eq. ' ') then
 EXEC SQL CONNECT TO sample
 else
 print *, 'Enter your password :'
 read 101, passwd

 EXEC SQL CONNECT TO sample USER :userid USING :passwd
 end if
 errloc = 'CONNECT'
 call checkerr (sqlca, errloc, *999)

* Employee A10030 is not included in the following select, because
* the lobeval program manipulates the record for A10030 so that
* it is not compatible with lobloc

 EXEC SQL DECLARE c1 CURSOR FOR
+ SELECT empno, resume FROM emp_resume
+ WHERE resume_format = 'ascii'
+ AND empno <> 'A00130'

 EXEC SQL OPEN c1
 errloc = 'OPEN CURSOR'
 call checkerr (sqlca, errloc, *999)

10 continue
```

```

 EXEC SQL FETCH c1 INTO :empnum, :resume :lobind
* 2
 if (sqlcode .ne. 0) goto 100
 if (lobind .lt. 0) goto 50

* The CLOB value exists.
* Locate the beginning of the
* "Department Information" section
 EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
+ INTO :di_begin_loc
 errloc = 'VALUES1'
 call checkerr (sqlca, errloc, *999)

* Locate the beginning of the
* "Education" section (end of Dept.Info)
 EXEC SQL VALUES (POSSTR(:resume, 'Education'))
+ INTO :di_end_loc
 errloc = 'VALUES2'
 call checkerr (sqlca, errloc, *999)

* Obtain ONLY the "Department Information"
* section by using SUBSTR
 EXEC SQL VALUES(SUBSTR(:resume, :di_begin_loc,
+ :di_end_loc - :di_begin_loc)) INTO :di_buffer
 errloc = 'VALUES3'
 call checkerr (sqlca, errloc, *999)

* Append the "Department Information"
* section to the :buffer variable
 EXEC SQL VALUES(:buffer || :di_buffer) INTO :buffer
 errloc = 'VALUES4'
 call checkerr (sqlca, errloc, *999)

 goto 10

50 print *, 'NULL LOB indicated'
 goto 10

100 print *, buffer

 EXEC SQL FREE LOCATOR :resume, :di_buffer
* 3
 errloc = 'FREE LOCATOR'
 call checkerr (sqlca, errloc, *999)

 EXEC SQL CLOSE c1
 errloc = 'CLOSE CURSOR'
 call checkerr (sqlca, errloc, *999)

 EXEC SQL CONNECT RESET
 errloc = 'CONNECT RESET'
 call checkerr (sqlca, errloc, *999)

999 stop
end

```

---

## Example: Deferring the Evaluation of a LOB Expression

There is no movement of the bytes of a LOB value until the assignment of a LOB expression to a target destination. This means that a LOB value locator used with string functions and operators can create an expression where the evaluation is postponed until the time of assignment. This is called *deferring evaluation* of a LOB expression.

In this example, a particular resume (empno = '000130') is sought within a table of resumes EMP\_RESUME. The Department Information section of the resume is copied, cut, and then appended to the end of the resume. This new resume will then be inserted into the EMP\_RESUME table. The original resume in this table remains unchanged.

Locators permit the assembly and examination of the new resume without actually moving or copying any bytes from the original resume. The movement of bytes does not happen until the final assignment; that is, the INSERT statement — and then only at the server.

Deferring evaluation gives DB2 an opportunity to increase LOB I/O performance. This occurs because the LOB function optimizer attempts to transform the LOB expressions into alternative expressions. These alternative expressions produce equivalent results but may also require fewer disk I/Os.

In summary, LOB locators are ideally suited for a number of programming scenarios:

1. When moving only a small part of a much larger LOB to a client program.
2. When the entire LOB cannot fit in the application's memory.
3. When the program needs a temporary LOB value from a LOB expression but does not need to save the result.
4. When performance is important (by deferring evaluation of LOB expressions).

## How the Sample LOBEVAL Program Works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. CLOB LOCATOR host variables are declared.
2. **Fetch the LOB value into the host variable LOCATOR.** A CURSOR and FETCH routine is used to obtain the location of a LOB field in the database to a host variable locator.
3. **LOB data is manipulated through the use of LOCATORS.** The next five SQL statements manipulate the LOB data without moving the actual data contained in the LOB field. This is done through the use of the LOB LOCATORS.
4. **LOB data is moved to the target destination.** The evaluation of the LOB assigned to the target destination is postponed until this SQL statement. The evaluation of this LOB statement has been deferred.
5. **Free the LOB LOCATORS.** The LOB LOCATORS used in this example are freed, releasing the locators from their previously associated values.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

**C**                    check\_error is redefined as CHECKERR and is located in the util.c file.

**COBOL**            CHECKERR is an external program named checkerr.cbl.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

## C Sample: LOBEVAL.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

 EXEC SQL BEGIN DECLARE SECTION; 1
 char userid[9];
 char passwd[19];
 long hv_start_deptinfo;
 long hv_start_educ;
 long hv_return_code;
 SQL TYPE IS CLOB(5K) hv_new_section_buffer;
 SQL TYPE IS CLOB_LOCATOR hv_doc_locator1;
 SQL TYPE IS CLOB_LOCATOR hv_doc_locator2;
 SQL TYPE IS CLOB_LOCATOR hv_doc_locator3;
 EXEC SQL END DECLARE SECTION;

#ifdef DB268K
 /* Before making any API calls for 68K environment,
 need to initial the Library Manager */
 InitLibraryManager(0,kCurrentZone,kNormalMemory);
 atexit(CleanupLibraryManager);
#endif

 printf("Sample C program: LOBEVAL\n");

 if (argc == 1) {
 EXEC SQL CONNECT TO sample;
 CHECKERR ("CONNECT TO SAMPLE");
 }
 else if (argc == 3) {
 strcpy (userid, argv[1]);
 strcpy (passwd, argv[2]);
 EXEC SQL CONNECT TO sample USER :userid USING :passwd;
 CHECKERR ("CONNECT TO SAMPLE");
 }
 else {
 printf ("USAGE: lobeval [userid passwd]");
 return 1;
 } /* endif */

 /* delete any instance of "A00130" from
 previous executions of this sample */
}
```

```

EXEC SQL DELETE FROM emp_resume WHERE empno = 'A00130';

/* Use a single row select to get the document */
EXEC SQL SELECT resume INTO :hv_doc_locator1 FROM emp_resume
WHERE empno = '000130' AND resume_format = 'ascii'; 2
CHECKERR ("SELECT");

/* Use the POSSTR function to locate the start of
sections "Department Information" & "Education" */
EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Department Information'))
INTO :hv_start_deptinfo; 3
CHECKERR ("VALUES1");

EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
INTO :hv_start_educ;
CHECKERR ("VALUES2");

/* Replace Department Information Section with nothing */
EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, 1, :hv_start_deptinfo -1)
|| SUBSTR (:hv_doc_locator1, :hv_start_educ))
INTO :hv_doc_locator2;
CHECKERR ("VALUES3");

/* Move Department Information Section
into the hv_new_section_buffer */
EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, :hv_start_deptinfo,
:hv_start_educ - :hv_start_deptinfo)) INTO :hv_new_section_buffer;
CHECKERR ("VALUES4");

/* Append our new section to the end (assume it has been filled in)
Effectively, this just moves the Department Information to the
bottom of the resume. */
EXEC SQL VALUES (:hv_doc_locator2 || :hv_new_section_buffer) INTO
:hv_doc_locator3;
CHECKERR ("VALUES5");

/* Store this resume section in the table. This is where the
LOB value bytes really move */
EXEC SQL INSERT INTO emp_resume VALUES ('A00130', 'ascii',
:hv_doc_locator3); 4
CHECKERR ("INSERT");

printf ("LOBEVAL completed\n");

/* free the locators */ 5
EXEC SQL FREE LOCATOR :hv_doc_locator1,
:hv_doc_locator2, : hv_doc_locator3;
CHECKERR ("FREE LOCATOR");

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : LOBEVAL.SQC */

```

## COBOL Sample: LOBEVAL.SQB

Identification Division.  
Program-ID. "lobeval".

Data Division.  
Working-Storage Section.

copy "sqlenv.cbl".  
copy "sql.cbl".  
copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC. **1**  
01 userid                   pic x(8).  
01 passwd.  
    49 passwd-length   pic s9(4) comp-5 value 0.  
    49 passwd-name   pic x(18).  
01 hv-start-deptinfo pic s9(9) comp-5.  
01 hv-start-educ   pic s9(9) comp-5.  
01 hv-return-code   pic s9(9) comp-5.  
01 hv-new-section-buffer USAGE IS SQL TYPE IS CLOB(5K).  
01 hv-doc-locator1   USAGE IS SQL TYPE IS CLOB-LOCATOR.  
01 hv-doc-locator2   USAGE IS SQL TYPE IS CLOB-LOCATOR.  
01 hv-doc-locator3   USAGE IS SQL TYPE IS CLOB-LOCATOR.  
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc               pic x(80).

Procedure Division.

Main Section.

display "Sample COBOL program: LOBEVAL".

\* Get database connection information.

display "Enter your user id (default none): "  
    with no advancing.  
accept userid.

if userid = spaces

EXEC SQL CONNECT TO sample END-EXEC

else

display "Enter your password : " with no advancing  
accept passwd-name.

\* Passwords in a CONNECT statement must be entered in a VARCHAR

\* format with the length of the input string.

inspect passwd-name tallying passwd-length for characters  
before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd  
END-EXEC.

move "CONNECT TO" to errloc.

call "checkerr" using SQLCA errloc.

\* Delete any instance of "A00130" from previous executions

EXEC SQL DELETE FROM emp\_resume  
WHERE empno = 'A00130' END-EXEC.

```

* use a single row select to get the document
EXEC SQL SELECT resume INTO :hv-doc-locator1
 FROM emp_resume
 WHERE empno = '000130'
 AND resume_format = 'ascii' END-EXEC.
move "SELECT" to errloc.
call "checkerr" using SQLCA errloc.

* use the POSSTR function to locate the start of sections
* "Department Information" & "Education"
EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
 'Department Information'))
 INTO :hv-start-deptinfo END-EXEC.
move "VALUES1" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL VALUES (POSSTR(:hv-doc-locator1,
 'Education')) INTO :hv-start-educ END-EXEC.
move "VALUES2" to errloc.
call "checkerr" using SQLCA errloc.

* replace Department Information section with nothing
EXEC SQL VALUES (SUBSTR(:hv-doc-locator1, 1,
 :hv-start-deptinfo - 1) ||
 SUBSTR(:hv-doc-locator1, :hv-start-educ))
 INTO :hv-doc-locator2 END-EXEC.
move "VALUES3" to errloc.
call "checkerr" using SQLCA errloc.

* move Department Information section into hv-new-section-buffer
EXEC SQL VALUES (SUBSTR(:hv-doc-locator1,
 :hv-start-deptinfo,
 :hv-start-educ - :hv-start-deptinfo))
 INTO :hv-new-section-buffer END-EXEC.
move "VALUES4" to errloc.
call "checkerr" using SQLCA errloc.

* Append the new section to the end (assume it has been filled)
* Effectively, this just moves the Dept Info to the bottom of
* the resume.
EXEC SQL VALUES (:hv-doc-locator2 ||
 :hv-new-section-buffer)
 INTO :hv-doc-locator3 END-EXEC.
move "VALUES5" to errloc.
call "checkerr" using SQLCA errloc.

* Store this resume in the table.
* This is where the LOB value bytes really move.
EXEC SQL INSERT INTO emp_resume
 VALUES ('A00130', 'ascii', :hv-doc-locator3)
 END-EXEC.
move "INSERT" to errloc.
call "checkerr" using SQLCA errloc.

```



```

display "LOBEVAL completed".

EXEC SQL FREE LOCATOR :hv-doc-locator1, :hv-doc-locator2,
 :hv-doc-locator3 END-EXEC.
move "FREE LOCATOR" to errloc.
call "checkerr" using SQLCA errloc.

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.

End-Prog.
stop run.

```

## Indicator Variables and LOB Locators

For normal host variables in an application program, when selecting a NULL value into a host variable, a negative value is assigned to the indicator variable signifying that the value is NULL. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Since a locator host variable itself can never be NULL, a negative indicator variable value indicates that the LOB value represented by the LOB locator is NULL. The NULL information is kept local to the client using the indicator variable value. The server does not track NULL values with valid locators.

---

## LOB File Reference Variables

File reference variables are similar to host variables except they are used to transfer data to and from client files, and not to and from memory buffers. A file reference variable represents (rather than contains) the file, just as a LOB locator represents (rather than contains) the LOB value. Database queries, updates, and inserts may use file reference variables to store, or to retrieve, single LOB values.

For very large objects, files are natural containers. In fact, it is likely that most LOBs begin as data stored in files on the client before they are moved to the database on the server. The use of file reference variables assists in moving LOB data. Programs use file reference variables to transfer LOB data from the client file directly to the database engine. The client application does not have to write utility routines to read and write files using host variables (which have size restrictions) to carry out the movement of LOB data.

**Note:** The file referenced by the file reference variable must be accessible from (but not necessarily resident on) the system on which the program runs. For a stored procedure, this would be the server.

A file reference variable has a data type of BLOB, CLOB, or DBCLOB. It is used either as the source of data (input) or as the target of data (output). The file reference variable may have a relative file name or a complete path name of the file (the latter is advised). The file name length is specified within the application program. The data length portion of the file reference variable is unused during input. During output, the data length is set by the application requestor code to the length of the new data written to the file.

When using file reference variables there are different options on both input and output. You must choose an action for the file by setting the `file_option` field in the file reference variable structure. Choices for assignment to the field covering both input and output values are shown below.

Values (shown for C) and options when using input file reference variables are as follows:

- **SQL\_FILE\_READ** (Regular file) — This is a file that can be open, read, and closed. DB2 determines the length of the data in the file (in bytes) when opening the file. DB2 then returns the length through the `data_length` field of the file reference variable structure. (The value for COBOL is `SQL-FILE-READ`, and for FORTRAN is `sql_file_read`.)

Values and options when using output file reference variables are as follows:

- **SQL\_FILE\_CREATE** (New file) — This option creates a new file. Should the file already exist, an error message is returned. (The value for COBOL is `SQL-FILE-CREATE`, and for FORTRAN is `sql_file_create`.)
- **SQL\_FILE\_OVERWRITE** (Overwrite file) — This option creates a new file if none already exists. If the file already exists, the new data overwrites the data in the file. (The value for COBOL is `SQL-FILE-OVERWRITE`, and for FORTRAN is `sql_file_overwrite`.)
- **SQL\_FILE\_APPEND** (Append file) — This option has the output appended to the file, if it exists. Otherwise, it creates a new file. (The value for COBOL is `SQL-FILE-APPEND`, and for FORTRAN is `sql_file_append`.)

**Notes:**

1. In an Extended UNIX Code (EUC) environment, the files to which DBCLOB file reference variables point are assumed to contain valid EUC characters appropriate for storage in a graphic column, and to never contain UCS-2 characters. For more information on DBCLOB files in an EUC environment, see “Considerations for DBCLOB Files” on page 424.
2. If a LOB file reference variable is used in an OPEN statement, the file associated with the LOB file reference variable must not be deleted until the cursor is closed.

For more information on file reference variables, refer to the *SQL Reference*.

---

## Example: Extracting a Document To a File

This program example shows how CLOB elements can be retrieved from a table into an external file.

### How the Sample LOBFILE Program Works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. A CLOB FILE REFERENCE host variable is declared.
2. **CLOB FILE REFERENCE host variable is set up.** The attributes of the FILE REFERENCE is set up. A file name without a fully declared path is, by default, placed in the current working directory.
3. **Select in to the CLOB FILE REFERENCE host variable.** The data from the resume field is selected into the filename referenced by the host variable.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

**C**                    check\_error is redefined as CHECKERR and is located in the util.c file.

**COBOL**            CHECKERR is an external program named checkerr.cbl

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

## C Sample: LOBFILE.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "util.h"

#ifdef DB268K
/* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

 EXEC SQL BEGIN DECLARE SECTION; 1
 SQL TYPE IS CLOB_FILE resume;
 short lobind;
 char userid[9];
 char passwd[19];
 EXEC SQL END DECLARE SECTION;

#ifdef DB268K
 /* Before making any API calls for 68K environment,
 need to initial the Library Manager */
 InitLibraryManager(0,kCurrentZone,kNormalMemory);
 atexit(CleanupLibraryManager);
#endif

 printf("Sample C program: LOBFILE\n");

 if (argc == 1) {
 EXEC SQL CONNECT TO sample;
 CHECKERR ("CONNECT TO SAMPLE");
 }
 else if (argc == 3) {
 strcpy (userid, argv[1]);
 strcpy (passwd, argv[2]);
 EXEC SQL CONNECT TO sample USER :userid USING :passwd;
 CHECKERR ("CONNECT TO SAMPLE");
 }
 else {
 printf ("\nUSAGE: lobfile [userid passwd]\n\n");
 return 1;
 } /* endif */

 strcpy (resume.name, "RESUME.TXT"); 2
 resume.name_length = strlen("RESUME.TXT");
 resume.file_options = SQL_FILE_OVERWRITE;

 EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume 3
 WHERE resume_format='ascii' AND empno='000130';
```

```

 if (lobind < 0) {
 printf ("NULL LOB indicated \n");
 } else {
 printf ("Resume for EMPNO 000130 is in file : RESUME.TXT\n");
 } /* endif */

 EXEC SQL CONNECT RESET;
 CHECKERR ("CONNECT RESET");
 return 0;
}
/* end of program : LOBFILE.SQC */

```

## COBOL Sample: LOBFILE.SQB

Identification Division.  
Program-ID. "lobfile".

Data Division.  
Working-Storage Section.

copy "sqlenv.cbl".  
copy "sql.cbl".  
copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC. **1**  
01 userid                   pic x(8).  
01 passwd.  
    49 passwd-length      pic s9(4) comp-5 value 0.  
    49 passwd-name        pic x(18).  
01 resume                 USAGE IS SQL TYPE IS CLOB-FILE.  
01 lobind                 pic s9(4) comp-5.  
EXEC SQL END DECLARE SECTION END-EXEC.  
  
77 errloc                 pic x(80).

Procedure Division.  
Main Section.

display "Sample COBOL program: LOBFILE".

\* Get database connection information.  
display "Enter your user id (default none): "  
    with no advancing.  
accept userid.  
  
if userid = spaces  
    EXEC SQL CONNECT TO sample END-EXEC  
else  
    display "Enter your password : " with no advancing  
    accept passwd-name.  
  
\* Passwords in a CONNECT statement must be entered in a VARCHAR  
\* format with the length of the input string.  
inspect passwd-name tallying passwd-length for characters  
    before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd  
END-EXEC.  
move "CONNECT TO" to errloc.  
call "checkerr" using SQLCA errloc.

move "RESUME.TXT" to resume-NAME. **2**  
move 10 to resume-NAME-LENGTH.  
move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

EXEC SQL SELECT resume INTO :resume :lobind **3**  
FROM emp\_resume  
WHERE resume\_format = 'ascii'  
AND empno = '000130' END-EXEC.

```
 if lobind less than 0 go to NULL-LOB-indicated.

 display "Resume for EMPNO 000130 is in file : RESUME.TXT".
 go to End-Main.

NULL-LOB-indicated.
 display "NULL LOB indicated".

End-Main.
 EXEC SQL CONNECT RESET END-EXEC.
 move "CONNECT RESET" to errloc.
 call "checkerr" using SQLCA errloc.
End-Prog.
 stop run.
```



---

## Example: Inserting Data Into a CLOB Column

In the path description of the following C program segment:

- `userid` represents the directory for one of your users.
- `dirname` represents a subdirectory name of “userid”.
- `filnam.1` can become the name of one of your documents that you wish to insert into the table.
- `clobtab` is the name of the table with the CLOB data type.

The following example shows how to insert data from a regular file referenced by `:hv_text_file` into a CLOB column (note that the path names used in the example are for UNIX-based systems):

```
strcpy(hv_text_file.name, "/u/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/u/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ; /* this is a 'regular' file */

EXEC SQL INSERT INTO CLOBTAB
VALUES(:hv_text_file);
```



---

## Chapter 9. User-Defined Functions (UDFs)

|                                           |     |                                                |     |
|-------------------------------------------|-----|------------------------------------------------|-----|
| What are UDFs? . . . . .                  | 255 | Example: AVG over a UDT . . . . .              | 264 |
| Why Use UDFs? . . . . .                   | 256 | Example: Counting . . . . .                    | 265 |
| UDF Concepts . . . . .                    | 258 | Example: Counting with an OLE                  |     |
| Implementing UDFs. . . . .                | 260 | automation object . . . . .                    | 265 |
| Writing UDFs . . . . .                    | 261 | Example: Table Function Returning              |     |
| Registering UDFs . . . . .                | 261 | Document IDs. . . . .                          | 266 |
| Examples of Registering UDFs . . . . .    | 261 | Using UDFs . . . . .                           | 267 |
| Example: Exponentiation . . . . .         | 261 | Referring to Functions . . . . .               | 267 |
| Example: String Search. . . . .           | 262 | Examples of Function Invocations . . . . .     | 268 |
| Example: BLOB String Search . . . . .     | 263 | Using Parameter Markers in Functions . . . . . | 269 |
| Example: String Search over UDT . . . . . | 263 | Using Qualified Function Reference . . . . .   | 269 |
| Example: External Function with UDT       |     | Using Unqualified Function Reference . . . . . | 270 |
| Parameter . . . . .                       | 264 | Summary of Function References . . . . .       | 271 |

---

### What are UDFs?

A user-defined function is a mechanism with which you can write your own extensions to SQL. The built-in functions supplied with DB2 are a useful set of functions, but they may not satisfy all of your requirements. Thus, you may need to extend SQL for the following reasons:

- **Customization.**

The function specific to your application does not exist in DB2. Whether the function is a simple transformation, a trivial calculation, or a complicated multivariate analysis, you can probably use a UDF to do the job.

- **Flexibility.**

The DB2 built-in function does not quite permit the variations that you wish to include in your application.

- **Standardization.**

Many of the programs at your site implement the same basic set of functions, but there are minor differences in all the implementations. Thus, you are unsure about the consistency of the results you receive. If you correctly implement these functions once, in a UDF, then all these programs can use the same implementation directly in SQL and provide consistent results.

- **Object-relational support.**

As discussed in “Chapter 10. User-defined Distinct Types (UDTs)” on page 275, UDTs can be very useful in extending the capability and increasing the safety of DB2. UDFs act as the *methods* for UDTs, by providing behavior and encapsulating the types.

---

## Why Use UDFs?

In writing DB2 applications, you have a choice when implementing desired actions or operations:

- As a UDF
- As a subroutine or function in your application.

Although it may seem easier to implement new operations as subroutines or functions in your application, there are good reasons why you should consider using UDFs:

- **Re-use.**

If the new operation is something of which other users or programs at your site can take advantage, then UDFs can help to reuse it. In addition, the function can be invoked directly in SQL wherever an expression can be used by any user of the database. The database will take care of many data type promotions of the function arguments automatically, for example DECIMAL to DOUBLE, allowing your function to be applied to different, but compatible data types.

It may seem easier to implement your new function as a normal function and then make it available to others for use in their programs, thereby avoiding the need to define the function to DB2. This requires that you inform all other interested application developers, and package the function effectively for their use. However, it ignores the interactive users like those who normally use the Command Line Processor (CLP) to access the database. CLP users cannot use your function unless it is a UDF in the database. This also applies to any other tools that use SQL (such as Lotus Approach), that do not get recompiled.

- **Performance.**

Invoking the UDF directly from the database engine instead of from your application can have a considerable performance advantage, particularly when the function may be used in the qualification of data for further processing. Consider a simple scenario where you want to process some data, provided you can meet some selection criteria which can be expressed as a function `SELECTION_CRITERIA()`. Your application could issue the following select statement:

```
SELECT A,B,C FROM T
```

When it receives each row, it runs `SELECTION_CRITERIA` against the data to decide if it is interested in processing the data further. Here, every row of table T must be passed back to the application. But if `SELECTION_CRITERIA()` is implemented as a UDF, your application can issue the following statement:

```
SELECT A,B,C FROM T WHERE SELECTION_CRITERIA(A,B) = 1
```

In this case, only the rows of interest are passed across the interface between the application and the database. For large tables, or for cases where `SELECTION_CRITERIA` supplies significant filtering, the performance improvement can be very significant.

Another case where a UDF can offer a performance benefit is when dealing with Large Objects (LOB). If you have a function whose purpose is to extract some information from a value of one of the LOB types, you can perform this extraction right on the database server and pass only the extracted value back to the application. This is more efficient than passing the entire LOB value back to the application and then performing the extraction. The performance value of packaging this function as a UDF could be enormous, depending on the particular situation. (Note that you can also extract a portion of a LOB by using a LOB locator. See “Example: Deferring the Evaluation of a LOB Expression” on page 239 for an example of a similar scenario.)

In addition, with table functions, you can very efficiently use relational operations and the power of SQL on data that resides outside a DB2 database (including non-relational data stores). A table function takes individual scalar values of different types and meanings as its arguments, and returns a table to the SQL statement that invokes it. You can write table functions that generate only the data in which you are interested, eliminating any unwanted rows or columns. For more information on table functions, including rules on where you can use them, refer to the *SQL Reference*.

- **Object Orientation.**

You can implement the behavior of a user-defined distinct type (UDT), also called *distinct type*, using a UDF. For more information on UDTs, see “Chapter 10. User-defined Distinct Types (UDTs)” on page 275. For additional details on UDTs and the important concept of *castability* discussed therein, refer to the *SQL Reference*. When you create a distinct type, you are automatically provided cast functions between the distinct type and its source type, and you may be provided comparison operators such as `=`, `>`, `<`, and so on, depending on the source type. You have to provide any additional behavior yourself. Because it is clearly desirable to keep the behavior of a distinct type in the database where all of the users of the distinct type can easily access it, UDFs can be used as the implementation mechanism.

For example, suppose you have a `BOAT` distinct type, defined over a one megabyte BLOB. The BLOB contains the various nautical specifications, and some drawings. You may wish to compare sizes of boats, and with a distinct type defined over a BLOB source type, you do not get the comparison operations automatically generated for you. You can implement a `BOAT_COMPARE` function which decides if one boat is bigger than

another based on a measure that you choose. These could be: displacement, length over all, metric tonnage, or another calculation based on the BOAT object. You create the BOAT\_COMPARE function as follows:

```
CREATE FUNCTION BOAT_COMPARE (BOAT, BOAT) RETURNS INTEGER ...
```

If your function returns 1 if the first BOAT is bigger, 2 if the second is bigger, and 0 if they are equal, you could use this function in your SQL code to compare boats. Suppose you create the following tables:

```
CREATE TABLE BOATS_INVENTORY (
 BOAT_ID CHAR(5),
 BOAT_TYPE VARCHAR(25),
 DESIGNER VARCHAR(40),
 OWNER VARCHAR(40),
 DESIGN_DATE DATE,
 SPEC BOAT,
 ...
)

CREATE TABLE MY_BOATS (
 BOAT_ID CHAR(5),
 BOAT_TYPE VARCHAR(25),
 DESIGNER VARCHAR(40),
 DESIGN_DATE DATE,
 ACQUIRE_DATE DATE,
 ACQUIRE_PRICE CANADIAN_DOLLAR,
 CURR_APPRAISL CANADIAN_DOLLAR,
 SPEC BOAT,
 ...
)
```

You can execute the following SQL SELECT statement:

```
SELECT INV.BOAT_ID, INV.BOAT_TYPE, INV.DESIGNER,
 INV.OWNER, INV.DESIGN_DATE
FROM BOATS_INVENTORY INV, MY_BOATS MY
WHERE MY.BOAT_ID = '19GCC'
AND BOAT_COMPARE(INV.SPEC, MY.SPEC) = 1
```

This simple example returns all the boats from BOATS\_INVENTORY that are bigger than a particular boat in MY\_BOATS. Note that the example only passes the rows of interest back to the application because the comparison occurs in the database server. In fact, it completely avoids passing any values of data type BOAT. This is a significant improvement in storage and performance as BOAT is based on a one megabyte BLOB data type.

---

## UDF Concepts

The following is a discussion of the important concepts you need to know prior to coding UDFs:

- Full name of a function

The full name of a function is <schema-name>.<function-name>. You can use this full name anywhere you refer to a function. For example:

SLICKO.BOAT\_COMPARE   SMITH.FOO   SYSIBM.SUBSTR   SYSFUN.FLOOR

However, you may also omit the <schema-name>., in which case, DB2 must identify the function to which you are referring. For example:

BOAT\_COMPARE   FOO   SUBSTR   FLOOR

- Function Path

The concept of *function path* is central to DB2's resolution of *unqualified* references that occur when you do not use the schema-name. For the use of function path in DDL statements that refer to functions, refer to the *SQL Reference*. The function path is an ordered list of schema names. It provides a set of schemas for resolving unqualified function references to UDFs as well as UDTs. In cases where a function reference matches functions in more than one schema in the path, the order of the schemas in the path is used to resolve this match. The function path is established by means of the FUNCPATH option on the precompile and bind commands for static SQL. The function path is set by the SET CURRENT FUNCTION PATH statement for dynamic SQL. The function path has the following default value:

"SYSIBM", "SYSFUN", "<ID>"

This applies to both static and dynamic SQL, where <ID> represents the current statement authorization ID.

- Overloaded function names

Function names can be *overloaded*, which means that multiple functions, even in the same schema, can have the same name. Two functions cannot, however, have the same *signature*, which can be defined to be the qualified function name concatenated with the defined data types of all the function parameters in the order in which they are defined. For an example of an overloaded function, see “Example: BLOB String Search” on page 263.

- Function selection algorithm

It is the *function selection algorithm* that takes into account the facts of overloading and function path to choose *the best fit* for every function reference, whether it is a qualified or an unqualified reference. Even references to the built-in functions and the functions (also IBM-supplied) in the SYSFUN schema are processed through the function selection algorithm.

- Types of function

Each user-defined function is classified as a *scalar*, *column* or *table* function. A *scalar function* returns a single value answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can either be external (coded in a programming language such as C), or sourced (using the implementation of an existing function).

A *column function* receives a set of like values (a column of data) and returns a single value answer from this set of values. These are also called *aggregating functions* in DB2. An example of a column function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF that is sourced on one of the built-in column functions can be defined. This is useful for distinct types. For example, if a distinct type SHOESIZE exists that is defined with base type INTEGER, you could define a UDF, AVG(SHOESIZE), as a column function sourced on the existing built-in column function, AVG(INTEGER).

A *table function* returns a table to the SQL statement that references it. A table function can only be referenced in the FROM clause of a SELECT statement. Such a function can be used to apply the SQL language to non-DB2 data, or to capture such data and put it into a DB2 table. For example, it could dynamically convert a file consisting of non-DB2 data into a table, or it could retrieve data from the World Wide Web or an operating system and tabularize it. A table function can only be an external function.

The concept of function path, the SET CURRENT FUNCTION PATH statement, and the function selection algorithm are discussed in detail in the *SQL Reference*. The FUNCPATH precompile and bind options are discussed in detail in the *Command Reference*.

For information about the concept of mapping UDFs and built-in functions to data source functions in a federated system, refer to the *SQL Reference*. For guidelines on creating such mappings, refer to “Invoking Data Source Functions” on page 482.

---

## Implementing UDFs

The process of implementing an external UDF requires the following steps:

1. Writing the UDF
2. Compiling the UDF
3. Linking the UDF
4. Debugging the UDF
5. Registering the UDF with DB2

After these steps are successfully completed, your UDF is ready for use in DML or DDL statements such as CREATE VIEW. The steps of writing and defining UDFs are discussed in the following sections, followed by a discussion on using UDFs. For information on compiling and linking UDFs, refer to the *Application Building Guide*. For information on debugging your UDF, see “Debugging your UDF” on page 371.



---

## Writing UDFs

You can find the details of how you write UDFs in “Chapter 11. Writing User-Defined Functions (UDFs)” on page 287. This includes the details on the interface between DB2 and UDFs, UDF coding considerations, UDF coding examples, and UDF debugging information. For information on the related tasks of compiling and linking your UDFs, refer to the *Application Building Guide*.

---

## Registering UDFs

You should register the UDF to DB2 after you have written and completely tested the actual code. Note that it is possible to define the UDF prior to actually writing it. However, to avoid any problems with running your UDF, you are encouraged to write and test it extensively before registering it. For information on testing your UDF, see “Debugging your UDF” on page 371.

Use the CREATE FUNCTION statement to define (or register) your UDF to DB2. You can find detailed explanations for this statement and its options in the *SQL Reference*.

---

## Examples of Registering UDFs

The examples which follow illustrate a variety of typical situations where UDFs can be registered.

Note that in these examples:

- The keyword or keyword/value specifications are always shown in the same order, for consistency of presentation and ease of understanding. In actually writing one of these CREATE FUNCTION statements, after the function name and the list of parameter data types, the specifications can appear in any order.
- The specifications in the EXTERNAL NAME clause are always shown for DB2 for AIX. You may need to make changes if you run these examples on non-UNIX platforms. For example, by converting all the slash (/) characters to back slash characters (\) and adding a drive letter such as C:, you have examples that are valid in OS/2 or Windows environments. Refer to the *SQL Reference* for a complete discussion of the EXTERNAL NAME clause.

### Example: Exponentiation

Suppose you have written an external UDF to perform exponentiation of floating point values, and wish to register it in the MATH schema. Assume

that you have DBADM authority. As you have tested the function extensively, and know that it does not represent any integrity exposure, you define it as NOT FENCED. By virtue of having DBADM authority, you possess the database authority, CREATE\_NOT\_FENCED, which is required to define the function as NOT FENCED.

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
 RETURNS DOUBLE
 EXTERNAL NAME '/common/math/exponent'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION
 NOT FENCED
```

In this example, the system uses the NOT NULL CALL default value. This is desirable since you want the result to be NULL if either argument is NULL. Since you do not require a scratchpad and no final call is necessary, the NO SCRATCHPAD and NO FINAL CALL default values are used. As there is no reason why EXPON cannot be parallel, the ALLOW PARALLELISM default value is used.

### Example: String Search

Your associate, Willie, has written a UDF to look for the existence of a given short string, passed as an argument, within a given CLOB value, which is also passed as an argument. The UDF returns the position of the string within the CLOB if it finds the string, or zero if it does not. Because you are concerned with database integrity for this function as you suspect the UDF is not fully tested, you define the function as FENCED.

Additionally, Willie has written the function to return a FLOAT result. Suppose you know that when it is used in SQL, it should always return an INTEGER. You can create the following function:

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
 RETURNS INTEGER
 CAST FROM FLOAT
 SPECIFIC "willie_find_feb95"
 EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION
 FENCED
```

Note that a CAST FROM clause is used to specify that the UDF body really returns a FLOAT value but you want to cast this to INTEGER before returning the value to the statement which used the UDF. As discussed in the *SQL*

*Reference*, the INTEGER built-in function can perform this cast for you. Also, you wish to provide your own specific name for the function and later reference it in DDL (see “Example: String Search over UDT”). Because the UDF was not written to handle NULL values, you use the NOT NULL CALL default value. And because there is no scratchpad, you use the NO SCRATCHPAD and NO FINAL CALL default values. As there is no reason why FINDSTRING cannot be parallel, the ALLOW PARALLELISM default value is used.

### Example: BLOB String Search

Because you want this function to work on BLOBs as well as on CLOBs, you define another FINDSTRING taking BLOB as the first parameter:

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
 RETURNS INTEGER
 CAST FROM FLOAT
 SPECIFIC "willie_fblob_feb95"
 EXTERNAL NAME '/u/willie/testfunc/testmod!findstr'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION
 FENCED
```

This example illustrates overloading of the UDF name, and shows that multiple UDFs can share the same body. Note that although a BLOB cannot be assigned to a CLOB, the same source code can be used. There is no programming problem in the above example as the programming interface for BLOB and CLOB between DB2 and UDF is the same; length followed by data. DB2 does not check if the UDF using a particular function body is in any way consistent with any other UDF using the same body.

### Example: String Search over UDT

This example is a continuation of the previous example. Say you are satisfied with the FINDSTRING functions from “Example: BLOB String Search”, but now you have defined a distinct type BOAT with source type BLOB. You also want FINDSTRING to operate on values having data type BOAT, so you create another FINDSTRING function. This function is sourced on the FINDSTRING which operates on BLOB values in “Example: BLOB String Search”. Note the further overloading of FINDSTRING in this example:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
 RETURNS INT
 SPECIFIC "slick_fboat_mar95"
 SOURCE SPECIFIC "willie_fblob_feb95"
```

Note that this FINDSTRING function has a different signature from the FINDSTRING functions in “Example: BLOB String Search” on page 263, so there is no problem overloading the name. You wish to provide our own specific name for possible later reference in DDL. Because you are using the SOURCE clause, you cannot use the EXTERNAL NAME clause or any of the related keywords specifying function attributes. These attributes are taken from the source function. Finally, observe that in identifying the source function you are using the specific function name explicitly provided in “Example: BLOB String Search” on page 263. Because this is an unqualified reference, the schema in which this source function resides must be in the function path, or the reference will not be resolved.

### Example: External Function with UDT Parameter

You have written another UDF to take a BOAT and examine its design attributes and generate a cost for the boat in Canadian dollars. Even though internally, the labor cost may be priced in euros, or Japanese yen, or US dollars, this function needs to generate the cost to build the boat in the required currency, Canadian dollars. This means it has to get current exchange rate information from an exchange rate web page, and the answer depends on the contents of the web page. This makes the function NOT DETERMINISTIC (or VARIANT).

```
CREATE FUNCTION BOAT_COST (BOAT)
 RETURNS INTEGER
 EXTERNAL NAME '/u/marine/funccdir/costs!boatcost'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 NOT DETERMINISTIC
 NO EXTERNAL ACTION
 FENCED
```

Observe that CAST FROM and SPECIFIC are not specified, but that NOT DETERMINISTIC is specified. Here again, FENCED is chosen for safety reasons.

### Example: AVG over a UDT

This example implements the AVG column function over the CANADIAN\_DOLLAR distinct type. See “Example: Money” on page 277 for the definition of CANADIAN\_DOLLAR. Strong typing prevents you from using the built-in AVG function on a distinct type. It turns out that the source type for CANADIAN\_DOLLAR was DECIMAL, and so you implement the AVG by sourcing it on the AVG(DECIMAL) built-in function. The ability to do this depends on being able to cast from DECIMAL to CANADIAN\_DOLLAR and vice versa, but since DECIMAL is the source type for CANADIAN\_DOLLAR you know these casts will work.

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
 RETURNS CANADIAN_DOLLAR
 SOURCE "SYSIBM".AVG(DECIMAL(9,2))
```

Note that in the `SOURCE` clause you have qualified the function name, just in case there might be some other `AVG` function lurking in your function path.

### Example: Counting

Your simple counting function returns a 1 the first time and increments the result by one each time it is called. This function takes no SQL arguments, and by definition it is a `NOT DETERMINISTIC` function since its answer varies from call to call. It uses the scratchpad to save the last value returned, and each time it is invoked it increments this value and returns it. You have rigorously tested this function, and possess `DBADM` authority on the database, so you will define it as `NOT FENCED`. (`DBADM` implies `CREATE_NOT_FENCED`.)

```
CREATE FUNCTION COUNTER ()
 RETURNS INT
 EXTERNAL NAME '/u/roberto/myfuncs/util!ctr'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 NOT DETERMINISTIC
 NOT FENCED
 SCRATCHPAD
 DISALLOW PARALLEL
```

Note that no parameter definitions are provided, just empty parentheses. The above function specifies `SCRATCHPAD`, and uses the default specification of `NO FINAL CALL`. In this case, as the size of the 100-byte scratchpad is sufficient, no storage has to be freed by means of a final call, and so `NO FINAL CALL` is specified. Since the `COUNTER` function requires that a single scratchpad be used to operate properly, `DISALLOW PARALLEL` is added to prevent DB2 from operating it in parallel. To see an implementation of this `COUNTER` function, refer to “Example: Counter” on page 353.

### Example: Counting with an OLE automation object

This example implements the previous counting example as an OLE (Object Linking and Embedding) automation object, `counter`, with an instance variable, `nbrOfInvoke`, to keep track of the number of invocations. Every time the UDF gets invoked, the `increment` method of the object increments the `nbrOfInvoke` instance variable and returns its current state. The automation object is registered in the Windows registry with the OLE programmatic identifier (progID) `bert.bcounter`.

```

CREATE FUNCTION bcounter ()
 RETURNS integer
 EXTERNAL NAME 'bert.bcounter!increment'
 LANGUAGE OLE
 PARAMETER STYLE DB2SQL
 SCRATCHPAD
 NOT DETERMINISTIC
 FENCED
 NULL CALL
 NO SQL
 NO EXTERNAL ACTION
 DISALLOW PARALLEL;

```

The implementation of the class counter is shown in “Example: Counter OLE Automation UDF in BASIC” on page 365 and in “Example: Counter OLE Automation UDF in C++” on page 367. For details of OLE support with DB2, see “Writing OLE Automation UDFs” on page 317.

### Example: Table Function Returning Document IDs

You have written a table function which returns a row consisting of a single document identifier column for each known document in your text management system which matches a given subject area (the first parameter) and contains the given string (second parameter). This UDF uses the functions of the text management system to quickly identify the documents:

```

CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
 RETURNS TABLE (DOC_ID CHAR(16))
 EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 DETERMINISTIC
 NO EXTERNAL ACTION
 NOT FENCED
 SCRATCHPAD
 NO FINAL CALL
 DISALLOW PARALLEL
 CARDINALITY 20

```

Within the context of a single session it will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH, including the column name DOC\_ID. FINAL CALL does not need to be specified for each table function. In addition, the DISALLOW PARALLEL keyword is added as table functions cannot operate in parallel. Although the size of the output from DOCMATCH is highly variable, CARDINALITY 20 is a representative value, and is specified to help the DB2 optimizer to make good decisions.

Typically this table function would be used in a join with the table containing the document text, as follows:

```
SELECT T.AUTHOR, T.DOCTEXT
FROM DOCS as T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN'S LEMMA')) as F
WHERE T.DOCID = F.DOC_ID
```

Note the special syntax (TABLE keyword) for specifying a table function in a FROM clause. In this invocation, the docmatch() table function returns a row containing the single column DOC\_ID for each mathematics document referencing Zorn's Lemma. These DOC\_ID values are joined to the master document table, retrieving the author's name and document text.

---

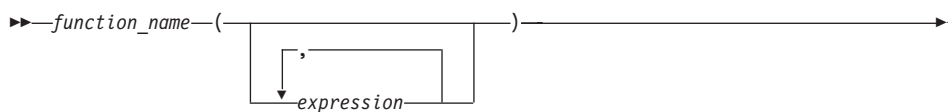
## Using UDFs

Scalar and column UDFs can be invoked within an SQL statement wherever an expression is valid (there are additional rules for all column functions that limit validity). Table UDFs can only be referenced in the FROM clause of a SELECT. The *SQL Reference* discusses all these contexts in detail. The discussion and examples used in this section focus on relatively simple SELECT statement contexts, but note that their use is not restricted to these contexts.

See “UDF Concepts” on page 258 for a summary of the use and importance of the *function path* and the *function selection* algorithm. You can find the details for both of these concepts in the *SQL Reference*. The resolution of any Data Manipulation Language (DML) reference to a function uses the function selection algorithm, so it is important to understand how it works.

## Referring to Functions

Each reference to a function, whether it is a UDF, or a built-in function, contains the following syntax:



In the above, `function_name` can be either an unqualified or a qualified function name, and the arguments can number from 0 to 90, and are expressions which may contain:

- A column name, qualified or unqualified
- A constant

- A host variable
- A special register
- A parameter marker. (For information on the limitations of parameter marker use, refer to the section in the *SQL Reference* that describes the rules for parameter markers.)

The position of the arguments is important and must conform to the function definition for the semantics to be correct. Both the position of the arguments and the function definition must conform to the function body itself. DB2 does not attempt to shuffle arguments to better match a function definition, and DB2 does not understand the semantics of the individual function parameters.

Use of column names in UDF argument expressions requires that the table references which contain the column have proper scope. For table functions referenced in a join, this means that for any argument which involves columns from another table or table function, that other table or table function must appear before the table function containing the reference, in the FROM clause. For a complete discussion of the rules for using columns in the arguments of table functions, refer to the *SQL Reference*.

## Examples of Function Invocations

Some valid examples of function invocations are:

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CHAR_COLUMN CONCAT USER, 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT FUNCTION PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
SYSFUN.FLOOR(AVG(EMP.SALARY))
SYSFUN.AVG(SYSFUN.FLOOR(EMP.SALARY))
SYSIBM.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
SQRT(SELECT SUM(length*length)
 FROM triangles
 WHERE id= 'J522'
 AND legtype <> 'HYP')
```

Note that if any of the above functions are table functions, the syntax to reference them is slightly different than presented above. For example, if PABLO.BLOOP is a table function, to properly reference it, use:

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```



## Using Parameter Markers in Functions

An important restriction involves parameter markers; you cannot simply code the following:

```
BLOOP(?)
```

As the function selection logic does not know what data type the argument may turn out to be, it cannot resolve the reference. You can use the CAST specification to provide a type for the parameter marker, for example INTEGER, and then the function selection logic can proceed:

```
BLOOP(CAST(? AS INTEGER))
```

## Using Qualified Function Reference

If you use a qualified function reference, you restrict DB2's search for a matching function to that schema. For example, you have the following statement:

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

Only the BLOOP functions in schema PABLO are considered. It does not matter that user SERGE has defined a BLOOP function, or whether or not there is a built-in BLOOP function. Now suppose that user PABLO has defined two BLOOP functions in his schema:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

BLOOP is thus overloaded within the PABLO schema, and the function selection algorithm would choose the best BLOOP, depending on the data type of the argument, column1. In this case, both of the PABLO.BLOOPS take numeric arguments, and if column1 is not one of the numeric types, the statement will fail. On the other hand if column1 is either SMALLINT or INTEGER, function selection will resolve to the first BLOOP, while if column1 is DECIMAL, DOUBLE, REAL, or BIGINT, the second BLOOP will be chosen.

Several points about this example:

1. It illustrates argument promotion. The first BLOOP is defined with an INTEGER parameter, yet you can pass it a SMALLINT argument. The function selection algorithm supports promotions among the built-in data types (for details, refer to the *SQL Reference*) and DB2 performs the appropriate data value conversions.
2. If for some reason you want to invoke the second BLOOP with a SMALLINT or INTEGER argument, you have to take an explicit action in your statement as follows:

```
SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
```

3. Alternatively, if you want to invoke the first BLOOP with a DECIMAL or DOUBLE argument, you have your choice of explicit actions, depending on your exact intent:

```
SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
SELECT PABLO.BLOOP(CEILING(COLUMN1)) FROM T
SELECT PABLO.BLOOP(INTEGER(ROUND(COLUMN1,0))) FROM T
```

You should investigate these other functions in the *SQL Reference*. The INTEGER function is a built-in function in the SYSIBM schema. The FLOOR, CEILING, and ROUND functions are UDFs shipped with DB2, which you can find in the SYSFUN schema along with many other useful functions.

## Using Unqualified Function Reference

If, instead of a qualified function reference, you use an unqualified function reference, DB2's search for a matching function normally uses the function path to qualify the reference. In the case of the DROP FUNCTION or COMMENT ON FUNCTION functions, the reference is qualified using the current authorization ID, if they are unqualified. Thus, *it is important that you know what your function path is, and what, if any, conflicting functions exist in the schemas of your current function path*. For example, suppose you are PABLO and your static SQL statement is as follows, where COLUMN1 is data type INTEGER:

```
SELECT BLOOP(COLUMN1) FROM T
```

You have created the two BLOOP functions cited in "Using Qualified Function Reference" on page 269, and you want and expect one of them to be chosen. If the following default function path is used, the first BLOOP is chosen (since column1 is INTEGER), if there is no conflicting BLOOP in SYSIBM or SYSFUN:

```
"SYSIBM", "SYSFUN", "PABLO"
```

However, suppose you have forgotten that you are using a script for precompiling and binding which you previously wrote for another purpose. In this script, you explicitly coded your FUNCPATH parameter to specify the following function path for another reason that does not apply to your current work:

```
"KATHY", "SYSIBM", "SYSFUN", "PABLO"
```

If Kathy has written a BLOOP function for her own purposes, the function selection could very well resolve to Kathy's function, and your statement would execute without error. You are not notified because DB2 assumes that you know what you are doing. It becomes your responsibility to identify the incorrect output from your statement and make the required correction.

## Summary of Function References

For both qualified and unqualified function references, the function selection algorithm looks at all the applicable functions, both built-in and user-defined, that have:

- The given name
- The same number of defined parameters as arguments in the function reference
- Each parameter identical to or promotable from the type of the corresponding argument.

(*Applicable functions* means *functions in the named schema* for a qualified reference, or *functions in the schemas of the function path* for an unqualified reference.) The algorithm looks for an exact match, or failing that, a best match among these functions. The current function path is used, in the case of an unqualified reference only, as the deciding factor if two identically good matches are found in different schemas. The details of the algorithm can be found in the *SQL Reference*.

An interesting feature, illustrated by the examples at the end of “Using Qualified Function Reference” on page 269, is *the fact that function references can be nested*, even references to the same function. This is generally true for built-in functions as well as UDFs; however, there are some limitations when column functions are involved.

Refining an earlier example:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

Now consider the following DML statement:

```
SELECT BLOOP(BLOOP(COLUMN1)) FROM T
```

If column1 is a DECIMAL or DOUBLE column, the inner BLOOP reference resolves to the second BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP resolves to the first BLOOP.

Alternatively, if column1 is a SMALLINT or INTEGER column, the inner bloop reference resolves to the first BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP also resolves to the first BLOOP. In this case, you are seeing nested references to the same function.

A few additional points important for function references are:

- By defining a function with the name of one of the SQL operators, you can actually invoke a UDF using *infix notation*. For example, suppose you can

attach some meaning to the "+" operator for values which have distinct type BOAT. You can define the following UDF:

```
CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
```

Then you can write the following valid SQL statement:

```
SELECT BOAT_COL1 + BOAT_COL2
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

But you can also write the equally valid statement:

```
SELECT "+"(BOAT_COL1, BOAT_COL2)
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

Note that you are not permitted to overload the built-in conditional operators such as >, =, LIKE, IN, and so on, in this way. See “Example: Integer Divide Operator” on page 344 for an example of a UDF which overloads the divide (/) operator.

- The function selection algorithm does not consider the context of the reference in resolving to a particular function. Look at these BLOOP functions, modified a bit from before:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
```

Now suppose you write the following SELECT statement:

```
SELECT 'ABCDEFGH' CONCAT BLOOP(SMALLINT_COL) FROM T
```

Because the best match, resolved using the SMALLINT argument, is the first BLOOP defined above, the second operand of the CONCAT resolves to data type INTEGER. The statement fails because CONCAT demands string arguments. If the first BLOOP was not present, the other BLOOP would be chosen and the statement execution would be successful.

Another type of contextual inconsistency that causes a statement to fail is if a given function reference resolves to a table function in a context that requires a scalar or column function. The reverse could also occur. A reference could resolve to a scalar or column function when a table function is necessary.

- UDFs can be defined with parameters or results having any of the LOB types: BLOB, CLOB, or DBCLOB. DB2 will materialize the entire LOB value in storage before invoking such a function, even if the source of the value is a *LOB locator* host variable. For example, consider the following fragment of a C language application:

```
EXEC SQL BEGIN DECLARE SECTION;
 SQL TYPE IS CLOB(150K) clob150K ; /* LOB host var */
 SQL TYPE IS CLOB_LOCATOR clob_locator1; /* LOB locator host var */
 char string[40]; /* string host var */
EXEC SQL END DECLARE SECTION;
```

Either host variable :clob150K or :clob\_locator1 is valid as an argument for a function whose corresponding parameter is defined as CLOB(500K). Thus, referring to the FINDSTRING defined in “Example: String Search” on page 262, both of the following are valid in the program:

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

- UDF parameters or results which have one of the LOB types can be created with the AS LOCATOR modifier. In this case, the entire LOB value is not materialized prior to invocation. Instead, a LOB LOCATOR is passed to the UDF, which can then use the special UDF APIs to manipulate the actual bytes of the LOB value (see “Using LOB Locators as UDF Parameters or Results” on page 335 for details).

You can also use this capability on UDF parameters or results which have a distinct type that is based on a LOB. This capability is limited to UDFs defined as not-fenced. Note that the argument to such a function can be any LOB value of the defined type; it does not have to be a host variable defined as one of the LOCATOR types. The use of host variable locators as arguments is completely orthogonal to the use of AS LOCATOR in UDF parameters and result definitions.

- UDFs can be defined with distinct types as parameters or as the result. (Earlier examples have illustrated this.) DB2 will pass the value to the UDF in the format of the source data type of the distinct type.

Distinct type values which originate in a host variable and which are used as arguments to a UDF which has its corresponding parameter defined as a distinct type, **must be explicitly cast to the distinct type by the user**. There is no host language type for distinct types. DB2’s strong typing necessitates this. Otherwise your results may be ambiguous. So, consider the BOAT distinct type which is defined over a BLOB, and consider the BOAT\_COST UDF from “Example: External Function with UDT Parameter” on page 264, which takes an object of type BOAT as its argument. In the following fragment of a C language application, the host variable :ship holds the BLOB value that is to be passed to the BOAT\_COST function:

```
EXEC SQL BEGIN DECLARE SECTION;
 SQL TYPE IS BLOB(150K) ship;
EXEC SQL END DECLARE SECTION;
```

Both of the following statements correctly resolve to the BOAT\_COST function, because both cast the :ship host variable to type BOAT:

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

If there are multiple BOAT distinct types in the database, or BOAT UDFs in other schema, you must exercise care with your function path. Otherwise your results may be ambiguous.

---

## Chapter 10. User-defined Distinct Types (UDTs)

|                                      |     |                                         |     |
|--------------------------------------|-----|-----------------------------------------|-----|
| Why Use UDTs? . . . . .              | 275 | Examples of Manipulating UDTs . . . . . | 279 |
| Defining a UDT . . . . .             | 276 | Example: Comparisons Between UDTs       |     |
| Resolving Unqualified UDTs . . . . . | 276 | and Constants. . . . .                  | 279 |
| Examples of Using CREATE DISTINCT    |     | Example: Casting Between UDTs . . . . . | 280 |
| TYPE. . . . .                        | 277 | Example: Comparisons Involving UDTs     | 281 |
| Example: Money . . . . .             | 277 | Example: Sourced UDFs Involving UDTs    | 282 |
| Example: Job Application . . . . .   | 277 | Example: Assignments Involving UDTs     | 282 |
| Defining Tables with UDTs . . . . .  | 277 | Example: Assignments in Dynamic SQL     | 283 |
| Example: Sales . . . . .             | 278 | Example: Assignments Involving          |     |
| Example: Application Forms . . . . . | 278 | Different UDTs . . . . .                | 283 |
| Manipulating UDTs . . . . .          | 279 | Example: Use of UDTs in UNION . . . . . | 284 |

---

### Why Use UDTs?

You can use data types that you have created, called user-defined distinct types (UDTs), in your DB2 applications. There are several benefits associated with UDTs:

1. **Extensibility.**

By defining new types, you can increase the set of types provided by DB2 to support your applications.

2. **Flexibility.**

You can specify any semantics and behavior for your new type by using user-defined functions (UDFs) to augment the diversity of the types available in the system. For more information on UDFs, see “Chapter 9. User-Defined Functions (UDFs)” on page 255.

3. **Consistent behavior.**

Strong typing insures that your UDTs will behave appropriately. It guarantees that only functions defined on your UDT can be applied to instances of the UDT.

4. **Encapsulation.**

The set of functions and operators that you can apply to UDTs defines the behavior of your UDTs. This provides flexibility in the implementation since running applications do not depend on the internal representation that you choose for your type.

5. **Performance.**

Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code used to implement built-in functions, comparison operators, indexes, etc. for built-in data types.

## 6. Foundation for object-oriented extensions.

UDTs are the foundation for most object-oriented features. They represent the most important step towards object-oriented extensions.

---

### Defining a UDT

UDTs, like other objects such as tables, indexes, and UDFs, need to be defined with a CREATE statement.

Use the CREATE DISTINCT TYPE statement to define your new UDT. Detailed explanations for the statement syntax and all its options are found in the *SQL Reference*.

For the CREATE DISTINCT TYPE statement, note that:

1. The name of the new UDT can be a qualified or an unqualified name. If it is qualified by a schema different from the authorization ID of the statement, you must have DBADM authority on the database.
2. The source type of the UDT is the type used by DB2 to internally represent the UDT. For this reason, it must be a built-in data type. Previously defined UDTs cannot be used as source types of other UDTs.
3. The WITH COMPARISONS clause is used to tell DB2 that functions to support the comparison operations on instances of the UDT should be generated by DB2. This clause is required if comparison operations are supported on the source type (for example, INTEGER and DATE) and is prohibited if comparison operations are not supported (for example, LONG VARCHAR and BLOB).

**Note:** As part of a UDT definition, DB2 always generates cast functions to:

- Cast from the UDT to the source type, using the standard name of the source type. For example, if you create a distinct type based on FLOAT, the cast function called DOUBLE is created.
- Cast from the source type to the UDT. Refer to the *SQL Reference* for a discussion of when additional casts to the UDTs are generated.

These functions are important for the manipulation of UDTs in queries.

---

### Resolving Unqualified UDTs

The function path is used to resolve any references to an unqualified type name or function, except if the type name or function is

- Created
- Dropped



- Commented on.

For information on how unqualified function references are resolved, see “Using Qualified Function Reference” on page 269.

---

## Examples of Using CREATE DISTINCT TYPE

The following are examples of using CREATE DISTINCT TYPE:

### Example: Money

Suppose you are writing applications that need to handle different currencies and wish to ensure that DB2 does not allow these currencies to be compared or manipulated directly with one another in queries. Remember that conversions are necessary whenever you want to compare values of different currencies. So you define as many UDTs as you need; one for each currency that you may need to represent:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL (9,2) WITH COMPARISONS
```

Note that you have to specify WITH COMPARISONS since comparison operators are supported on DECIMAL (9,2).

### Example: Job Application

Suppose you would like to keep the form filled by applicants to your company in a DB2 table and you are going to use functions to extract the information from these forms. Because these functions cannot be applied to regular character strings (because they are certainly not able to find the information they are supposed to return), you define a UDT to represent the filled forms:

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

Because DB2 does not support comparisons on CLOBs, you do not specify the clause WITH COMPARISONS. You have specified a schema name different from your authorization ID since you have DBADM authority, and you would like to keep all UDTs and UDFs dealing with applicant forms in the same schema.

---

## Defining Tables with UDTs

After you have defined several UDTs, you can start defining tables with columns whose types are UDTs. Following are examples using CREATE TABLE:

## Example: Sales

Suppose you want to define tables to keep your company's sales in different countries as follows:

```
CREATE TABLE US_SALES
(PRODUCT_ITEM INTEGER,
 MONTH INTEGER CHECK (MONTH BETWEEN 1 AND 12),
 YEAR INTEGER CHECK (YEAR > 1985),
 TOTAL US_DOLLAR)

CREATE TABLE CANADIAN_SALES
(PRODUCT_ITEM INTEGER,
 MONTH INTEGER CHECK (MONTH BETWEEN 1 AND 12),
 YEAR INTEGER CHECK (YEAR > 1985),
 TOTAL CANADIAN_DOLLAR)

CREATE TABLE GERMAN_SALES
(PRODUCT_ITEM INTEGER,
 MONTH INTEGER CHECK (MONTH BETWEEN 1 AND 12),
 YEAR INTEGER CHECK (YEAR > 1985),
 TOTAL EURO)
```

The UDTs in the above examples are created using the same CREATE DISTINCT TYPE statements in “Example: Money” on page 277. Note that the above examples use check constraints. For information on check constraints refer to the *SQL Reference*.

## Example: Application Forms

Suppose you need to define a table where you keep the forms filled out by applicants as follows:

```
CREATE TABLE APPLICATIONS
(ID SYSIBM.INTEGER,
 NAME VARCHAR (30),
 APPLICATION_DATE SYSIBM.DATE,
 FORM PERSONAL.APPLICATION_FORM)
```

You have fully qualified the UDT name because its qualifier is not the same as your authorization ID and you have not changed the default function path. Remember that whenever type and function names are not fully qualified, DB2 searches through the schemas listed in the current function path and looks for a type or function name matching the given unqualified name. Because SYSIBM is always considered (if it has been omitted) in the current function path, you can omit the qualification of built-in data types. For example, you can execute SET CURRENT FUNCTION PATH = cheryl and the value of the current function path special register will be "CHERYL", and does not include "SYSIBM". Now, if CHERYL.INTEGER type is not defined, the statement CREATE TABLE FOO(COL1 INTEGER) still succeeds because SYSIBM is always considered as COL1 is of type SYSIBM.INTEGER.

You are, however, allowed to fully qualify the built-in data types if you wish to do so. Details about the use of the current function path are discussed in the *SQL Reference*.

---

## Manipulating UDTs

One of the most important concepts associated with UDTs is *strong typing*. Strong typing guarantees that only functions and operators defined on the UDT can be applied to its instances.

Strong typing is important to ensure that the instances of your UDTs are correct. For example, if you have defined a function to convert US dollars to Canadian dollars according to the current exchange rate, you do not want this same function to be used to convert euros to Canadian dollars because it will certainly return the wrong amount.

As a consequence of strong typing, DB2 does not allow you to write queries that compare, for example, UDT instances with instances of the UDT source type. For the same reason, DB2 will not let you apply functions defined on other types to UDTs. If you want to compare instances of UDTs with instances of another type, you have to cast the instances of one or the other type. In the same sense, you have to cast the UDT instance to the type of the parameter of a function that is not defined on a UDT if you want to apply this function to a UDT instance.

---

## Examples of Manipulating UDTs

The following are examples of manipulating UDTs:

- Example: Comparisons Between UDTs and Constants
- Example: Casting Between UDTs
- Example: Comparisons Involving UDTs
- Example: Sourced UDFs Involving UDTs
- Example: Assignments Involving UDTs
- Example: Assignments in Dynamic SQL
- Example: Assignments Involving Different UDTs
- Example: Use of UDTs in UNION

### Example: Comparisons Between UDTs and Constants

Suppose you want to know which products sold more than US \$100 000.00 in the US in the month of July, 1999 (7/99).

```

SELECT PRODUCT_ITEM
FROM US_SALES
WHERE TOTAL > US_DOLLAR (100000)
AND month = 7
AND year = 1999

```

Because you cannot compare US dollars with instances of the source type of US dollars (that is, DECIMAL) directly, you have used the cast function provided by DB2 to cast from DECIMAL to US dollars. You can also use the other cast function provided by DB2 (that is, the one to cast from US dollars to DECIMAL) and cast the column total to DECIMAL. Either way you decide to cast, from or to the UDT, you can use the cast specification notation to perform the casting, or the functional notation. That is, you could have written the above query as:

```

SELECT PRODUCT_ITEM
FROM US_SALES
WHERE TOTAL > CAST (100000 AS us_dollar)
AND MONTH = 7
AND YEAR = 1999

```

### Example: Casting Between UDTs

Suppose you want to define a UDF that converts Canadian dollars to U.S. dollars. Suppose you can obtain the current exchange rate from a file managed outside of DB2. You would then define a UDF that obtains a value in Canadian dollars, accesses the exchange rate file, and returns the corresponding amount in U.S. dollars.

At first glance, such a UDF may appear easy to write. However, C does not support DECIMAL values. The UDTs representing different currencies have been defined as DECIMAL. Your UDF will need to receive and return DOUBLE values, since this is the only data type provided by C that allows the representation of a DECIMAL value without losing the decimal precision. Thus, your UDF should be defined as follows:

```

CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
EXTERNAL NAME '/u/finance/funmdir/currencies!cdn2us'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
NOT DETERMINISTIC
NO EXTERNAL ACTION
FENCED

```

The exchange rate between Canadian and U.S. dollars may change between two invocations of the UDF, so you declare it as NOT DETERMINISTIC.

The question now is, how do you pass Canadian dollars to this UDF and get U.S. dollars from it? The Canadian dollars must be cast to DECIMAL values.

The DECIMAL values must be cast to DOUBLE. You also need to have the returned DOUBLE value cast to DECIMAL and the DECIMAL value cast to U.S. dollars.

Such casts are performed automatically by DB2 anytime you define sourced UDFs, whose parameter and return type do not exactly match the parameter and return type of the source function. Therefore, you need to define two sourced UDFs. The first brings the DOUBLE values to a DECIMAL representation. The second brings the DECIMAL values to the UDT. That is, you define the following:

```
CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
 SOURCE CDN_TO_US_DOUBLE (DOUBLE)

CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
 SOURCE CDN_TO_US_DEC (DECIMAL())
```

Note that an invocation of the US\_DOLLAR function as in US\_DOLLAR(C1), where C1 is a column whose type is Canadian dollars, has the same effect as invoking:

```
US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1)))))
```

That is, C1 (in Canadian dollars) is cast to decimal which in turn is cast to a double value that is passed to the CDN\_TO\_US\_DOUBLE function. This function accesses the exchange rate file and returns a double value (representing the amount in U.S. dollars) that is cast to decimal, and then to U.S. dollars.

A function to convert euros to U.S. dollars would be similar to the example above:

```
CREATE FUNCTION EURO_TO_US_DOUBL(DOUBLE)
 RETURNS DOUBLE
 EXTERNAL NAME '/u/finance/fncdir/currencies!euro2us'
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 NO SQL
 NOT DETERMINISTIC
 NO EXTERNAL ACTION
 FENCED

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
 RETURNS DECIMAL(9,2)
 SOURCE EURO_TO_US_DOUBL (DOUBLE)

CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
 SOURCE EURO_TO_US_DEC (DECIMAL())
```

### **Example: Comparisons Involving UDTs**

Suppose you want to know which products sold more in the US than in Canada and Germany for the month of July, 1999 (7/1999):

```

SELECT US.PRODUCT_ITEM, US.TOTAL
 FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
 WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
 AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
 AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
 AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
 AND US.MONTH = 7
 AND US.YEAR = 1999
 AND CDN.MONTH = 7
 AND CDN.YEAR = 1999
 AND GERMAN.MONTH = 7
 AND GERMAN.YEAR = 1999

```

Because you cannot directly compare US dollars with Canadian dollars or euros, you use the UDF to cast the amount in Canadian dollars to US dollars, and the UDF to cast the amount in euros to US dollars. You cannot cast them all to DECIMAL and compare the converted DECIMAL values because the amounts are not monetarily comparable. That is, the amounts are not in the same currency.

### Example: Sourced UDFs Involving UDTs

Suppose you have defined a sourced UDF on the built-in SUM function to support SUM on euros:

```

CREATE FUNCTION SUM (EUROS)
 RETURNS EUROS
 SOURCE SYSIBM.SUM (DECIMAL())

```

You want to know the total of sales in Germany for each product in the year of 1994. You would like to obtain the total sales in US dollars:

```

SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
 FROM GERMAN_SALES
 WHERE YEAR = 1994
 GROUP BY PRODUCT_ITEM

```

You could not write SUM (us\_dollar (total)), unless you had defined a SUM function on US dollar in a manner similar to the above.

### Example: Assignments Involving UDTs

Suppose you want to store the form filled by a new applicant into the database. You have defined a host variable containing the character string value used to represent the filled form:

```

EXEC SQL BEGIN DECLARE SECTION;
 SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */

```

```
INSERT INTO APPLICATIONS
VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

You do not explicitly invoke the cast function to convert the character string to the UDT `personal.application_form` because DB2 lets you assign instances of the source type of a UDT to targets having that UDT.

### Example: Assignments in Dynamic SQL

If you want to use the same statement given in “Example: Assignments Involving UDTs” on page 282 in dynamic SQL, you can use parameter markers as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
 long id;
 char name[30];
 SQL TYPE IS CLOB(32K) form;
 char command[80];
EXEC SQL END DECLARE SECTION;

/* Code to fill host variables */

strcpy(command, "INSERT INTO APPLICATIONS VALUES");
strcat(command, "(?, ?, CURRENT DATE, CAST (? AS CLOB(32K)))");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;
```

You made use of DB2’s cast specification to tell DB2 that the type of the parameter marker is CLOB(32K), a type that is assignable to the UDT column. Remember that you cannot declare a host variable of a UDT type, since host languages do not support UDTs. Therefore, you cannot specify that the type of a parameter marker is a UDT.

### Example: Assignments Involving Different UDTs

Suppose you have defined two sourced UDFs on the built-in SUM function to support SUM on US and Canadian dollars, similar to the UDF sourced on euros in “Example: Sourced UDFs Involving UDTs” on page 282:

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)
 RETURNS CANADIAN_DOLLAR
 SOURCE SYSIBM.SUM (DECIMAL())

CREATE FUNCTION SUM (US_DOLLAR)
 RETURNS US_DOLLAR
 SOURCE SYSIBM.SUM (DECIMAL())
```

Now suppose your supervisor requests that you maintain the annual total sales in US dollars of each product and in each country, in separate tables:

```
CREATE TABLE US_SALES_94
 (PRODUCT_ITEM INTEGER,
 TOTAL US_DOLLAR)

CREATE TABLE GERMAN_SALES_94
 (PRODUCT_ITEM INTEGER,
 TOTAL US_DOLLAR)

CREATE TABLE CANADIAN_SALES_94
 (PRODUCT_ITEM INTEGER,
 TOTAL US_DOLLAR)

INSERT INTO US_SALES_94
 SELECT PRODUCT_ITEM, SUM (TOTAL)
 FROM US_SALES
 WHERE YEAR = 1994
 GROUP BY PRODUCT_ITEM

INSERT INTO GERMAN_SALES_94
 SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
 FROM GERMAN_SALES
 WHERE YEAR = 1994
 GROUP BY PRODUCT_ITEM

INSERT INTO CANADIAN_SALES_94
 SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
 FROM CANADIAN_SALES
 WHERE YEAR = 1994
 GROUP BY PRODUCT_ITEM
```

You explicitly cast the amounts in Canadian dollars and euros to US dollars since different UDTs are not directly assignable to each other. You cannot use the cast specification syntax because UDTs can only be cast to their own source type.

### Example: Use of UDTs in UNION

Suppose you would like to provide your American users with a view containing all the sales of every product of your company:

```
CREATE VIEW ALL_SALES AS
 SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
 FROM US_SALES
 UNION
 SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
 FROM CANADIAN_SALES
 UNION
 SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
 FROM GERMAN_SALES
```



You cast Canadian dollars to US dollars and euros to US dollars because UDTs are union compatible only with the same UDT. Note that you have to use the functional notation to cast between UDTs since the cast specification only lets you cast between UDTs and their source types.



---

## Chapter 11. Writing User-Defined Functions (UDFs)

|                                           |     |                                            |     |
|-------------------------------------------|-----|--------------------------------------------|-----|
| Description . . . . .                     | 287 | Defining a Server Name for an OLE DB       |     |
| Interface between DB2 and a UDF . . . . . | 288 | Provider . . . . .                         | 328 |
| The Arguments Passed from DB2 to a        |     | Defining a User Mapping . . . . .          | 328 |
| UDF . . . . .                             | 288 | Supported OLE DB Data Types . . . . .      | 329 |
| Summary of UDF Argument Use . . . . .     | 301 | Scratchpad Considerations . . . . .        | 331 |
| How the SQL Data Types are Passed to a    |     | Table Function Considerations . . . . .    | 333 |
| UDF . . . . .                             | 303 | Table Function Error Processing . . . . .  | 334 |
| The UDF Include File: sqludf.h . . . . .  | 311 | Scalar Function Error Processing . . . . . | 334 |
| Creating and Using Java User-Defined      |     | Using LOB Locators as UDF Parameters or    |     |
| Functions . . . . .                       | 312 | Results . . . . .                          | 335 |
| Coding a Java UDF . . . . .               | 313 | Scenarios for Using LOB Locators . . . . . | 339 |
| Changing How a Java UDF Runs . . . . .    | 314 | Other Coding Considerations . . . . .      | 340 |
| Table Function Execution Model for Java   | 315 | Hints and Tips . . . . .                   | 340 |
| Writing OLE Automation UDFs . . . . .     | 317 | UDF Restrictions and Caveats . . . . .     | 342 |
| Creating and Registering OLE              |     | Examples of UDF Code . . . . .             | 344 |
| Automation UDFs . . . . .                 | 318 | Example: Integer Divide Operator . . . . . | 344 |
| Object Instance and Scratchpad            |     | Example: Fold the CLOB, Find the Vowel     | 348 |
| Considerations . . . . .                  | 319 | Example: Counter . . . . .                 | 353 |
| How the SQL Data Types are Passed to      |     | Example: Weather Table Function . . . . .  | 355 |
| an OLE Automation UDF . . . . .           | 319 | Example: Function using LOB locators       | 362 |
| Implementing OLE Automation UDFs in       |     | Example: Counter OLE Automation UDF        |     |
| BASIC and C++ . . . . .                   | 320 | in BASIC . . . . .                         | 365 |
| OLE Automation UDFs in BASIC              | 321 | Example: Counter OLE Automation UDF        |     |
| OLE Automation UDFs in C++ . . . . .      | 321 | in C++ . . . . .                           | 367 |
| OLE DB Table Functions . . . . .          | 324 | Example: Mail OLE Automation Table         |     |
| Creating an OLE DB Table Function         | 325 | Function in BASIC . . . . .                | 369 |
| Fully Qualified Rowset Names . . . . .    | 327 | Debugging your UDF . . . . .               | 371 |

---

### Description

After a preliminary discussion on the interface between DB2 and a UDF, the remaining discussion concerns how you implement UDFs. The information on writing the UDF emphasizes the presence or absence of a scratchpad as one of the primary considerations.

Some general considerations in using this section are:

- Important material on defining and using UDFs is presented in “Chapter 9. User-Defined Functions (UDFs)” on page 255, and is not repeated here. This discussion concentrates on how you implement a UDF.
- To implement an *external UDF* written in C, C++ or Java, you must perform the following steps:
  - Write the UDF

- Compile the UDF
- Link the UDF
- Register the UDF with a CREATE FUNCTION statement
- Test and debug the UDF

You can find information on compiling and linking UDFs in the *Application Building Guide*.

- You can invoke your UDF using OLE (Object Linking and Embedding) as described in “Writing OLE Automation UDFs” on page 317.
- You can define an *OLE DB table function*, which is a function that returns a table from an OLE DB data source, with just a CREATE FUNCTION statement. For more information on OLE DB table functions, see “OLE DB Table Functions” on page 324.

Note that a *sourced UDF*, which is different from an external UDF, does not require an implementation in the form of a separate piece of code. Such a UDF uses the same implementation as its source function, along with many of its other attributes.

---

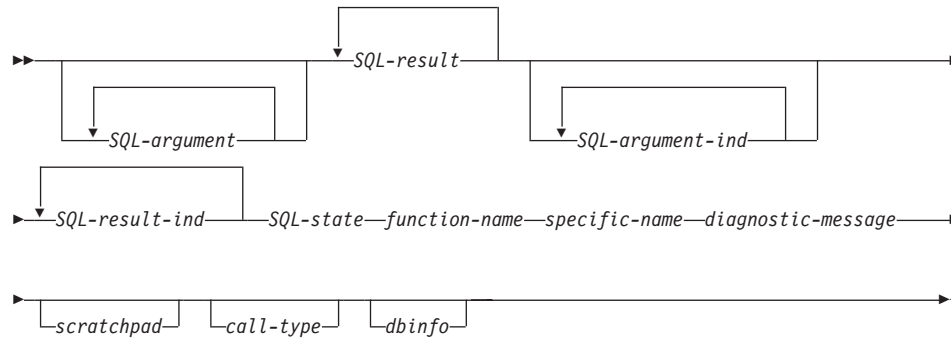
## Interface between DB2 and a UDF

This section discusses some of the details of the interface between DB2 and a UDF, and discusses the `sqludf.h` include file which makes the interface manageable. This include file only applies to C and C++ UDFs. For information on coding UDFs in Java, see “Coding a Java UDF” on page 313.

### The Arguments Passed from DB2 to a UDF

In addition to the SQL arguments which are specified in the DML reference to the function, DB2 passes additional arguments to the external UDF. For C and C++, all of these arguments are passed in the order shown in “Passing Arguments to a UDF” on page 289. Java UDFs take only the *SQL-argument* and *SQL-result* arguments, but can call extra methods to access the other information. Java UDFs have the same restrictions on the resulting *SQL-state* and *diagnostic-message* arguments documented below. For information on coding UDFs in Java, see “Coding a Java UDF” on page 313.

## Syntax for Passing Arguments to a UDF



**Note:** Each of the above arguments passed to the external function is a pointer to the value, and not the actual value.

The arguments are described as follows:

### *SQL-argument*

This argument is set by DB2 before calling the UDF. This value repeats  $n$  times, where  $n$  is the number of arguments specified in the function reference. The value of each of these arguments is taken from the expression specified in the function invocation. It is expressed in the data type of the corresponding parameter definition in the CREATE FUNCTION statement. How these data types map to C language constructs is described in “How the SQL Data Types are Passed to a UDF” on page 303.

### *SQL-result*

This argument is set by the UDF before returning to DB2. For scalar functions there is exactly one SQL-result. For table functions there is one SQL-result for each result column of the function defined in the RETURNS TABLE clause of the CREATE FUNCTION statement. They correspond by position to the columns defined in the RETURNS TABLE clause. That is, the first SQL-result argument corresponds to the first column defined in the RETURNS TABLE clause, and so on.

For both scalar functions and table functions, DB2 allocates the buffer and passes its address to the UDF. The UDF puts each result value into the buffer. Enough buffer space is allocated by DB2 to contain the value expressed in the data type. For scalar functions, this data type is defined in the CAST FROM clause, if it is present, or in the RETURNS clause, if no CAST FROM clause is present. For table functions, the data types are defined in the RETURNS TABLE(...) clause. For

information on how these types map to C language constructs, see “How the SQL Data Types are Passed to a UDF” on page 303.

Note that for table functions, DB2 defines a performance optimization where every defined column does not have to be returned to DB2. If you write your UDF to take advantage of this feature, it returns only the columns required by the statement referencing the table function.

For example, consider a CREATE FUNCTION statement for a table function defined with 100 result columns. If a given statement referencing the function is only interested in two of them, this optimization enables the UDF to return only those two columns for each row and not spend time on the other 98 columns. See the *dbinfo* argument below for more information on this optimization.

For each value returned, (that is, a single value for a scalar function, and in general, multiple values for a table function), the UDF code should not return more bytes than is required for the data type and length of the result. DB2 will attempt to determine if the UDF body has written beyond the end of the result buffer by a few bytes, returning SQLCODE -450 (SQLSTATE 39501). However, a major overwrite by the UDF that DB2 does not detect can cause unpredictable results or an abnormal termination.

#### *SQL-argument-ind*

This argument is set by DB2 before calling the UDF. It can be used by the UDF to determine if the corresponding *SQL-argument* is null or not. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* (described above). It contains one of the following values:

- 0        The argument is present and not null.
- 1       The argument is present and its value is null.

If the function is defined with NOT NULL CALL, the UDF body does not need to check for a null value. However, if it is defined with NULL CALL, any argument can be NULL and the UDF should check it.

The indicator takes the form of a SMALLINT value, and this can be defined in your UDF as described in “How the SQL Data Types are Passed to a UDF” on page 303.

#### *SQL-result-ind*

This argument is set by the UDF before returning to DB2. There is one of these for each *SQL-result* argument.

This argument is used by the UDF to signal if the particular result value is null:

**0 or positive**

The result is not null

**negative**

The result is the null value. For more information, see “Interpreting Negative SQL-result-ind Values”.

**Interpreting Negative SQL-result-ind Values:**

DB2 treats the function result as null (-2) if the following is true:

- The database configuration parameter DFT\_SQLMATHWARN is 'YES'
- One of the input arguments is a null because of an arithmetic error
- The SQL-result-ind is negative.

This is also true if you define the function with the NOT NULL CALL option.

Even if the function is defined with NOT NULL CALL, the UDF body must set the indicator of the result. For example, a divide function could set the result to null when the denominator is zero.

The indicator takes the form of a SMALLINT value, and this can be defined in your UDF as described in “How the SQL Data Types are Passed to a UDF” on page 303.

If the UDF takes advantage of table function optimization using the RESULT column list, then only the indicators corresponding to the required columns need be set.

***SQL-state***

This argument is set by the UDF before returning to DB2. It takes the form of a CHAR(5) value. Ensure that the argument definition in the UDF is appropriate for a CHAR(5) as described in “How the SQL Data Types are Passed to a UDF” on page 303, and can be used by the UDF to signal warning or error conditions. It contains the value '00000', when the function is called. The UDF can set the value to the following:

- 00000** The function code did not detect any warning or error situations.
- 01Hxx** The function code detected a warning situation. This results in a SQL warning, SQLCODE +462 (SQLSTATE 01Hxx ). Here 'xx' is any string.
- 02000** Only valid for the FETCH call to table functions, it means that there are no more rows in the table.

**38502** A special value for the case where the UDF body attempted to issue an SQL call and received an error, SQLCODE -487 (SQLSTATE 38502). because SQL is not allowed in UDFs), and chose to pass this same error back through to DB2.

**Any other 38xxx**

The function code detected an error situation. It results in a SQL error, SQLCODE -443 (SQLSTATE 38xxx). Here 'xxx' is any string. Do not use 380xx through 384xx because those values are reserved by the draft extensions to the SQL92 international standard, or 385xx because those values are reserved by IBM.

Any other value is treated as an error situation resulting in SQLCODE -463 (SQLSTATE 39001).

*function-name*

This argument is set by DB2 before calling the UDF. It is the qualified function name, passed from DB2 to the UDF code. This variable takes the form of a VARCHAR(27) value. Ensure that the argument definition in the UDF is appropriate for a VARCHAR(27). See “How the SQL Data Types are Passed to a UDF” on page 303 for more information.

The form of the function name that is passed is:

*<schema-name>.<function-name>*

The parts are separated by a period. Two examples are:

PABLO.BLOOP      WILLIE.FINDSTRING

This form enables you to use the same UDF body for multiple external functions, and still differentiate between the functions when it is invoked.

**Note:** Although it is possible to include the period in object names and schema names, it is not recommended. For example, if a function, rotate is in a schema, obj.op, the function name that is returned is obj.op.rotate, and it is not obvious if the schema name is obj or obj.op.

*specific-name*

This argument is set by DB2 before calling the UDF. It is the specific name of the function passed from DB2 to the UDF code. This variable takes the form of a VARCHAR(18) value. Ensure that the argument definition in the UDF is appropriate for a VARCHAR(18). See “How the SQL Data Types are Passed to a UDF” on page 303 for more information. Two examples are:



This first value is provided by the user in his CREATE FUNCTION statement. The second value is generated by DB2 from the current timestamp if the user does not specify a value.

As with the *function-name* argument, the reason for passing this value is to give the UDF the means of distinguishing exactly which specific function is invoking it.

*diagnostic-message*

This argument is set by the UDF before returning to DB2. The UDF can use this argument to insert a message text in a DB2 message. It takes the form of a VARCHAR(70) value. Ensure that the argument definition in the UDF is appropriate for a VARCHAR(70). See “How the SQL Data Types are Passed to a UDF” on page 303 for more information.

When the UDF returns either an error or a warning, using the *SQL-state* argument described above, it can include descriptive information here. DB2 includes this information as a token in its message.

DB2 sets the first character to null before calling the UDF. Upon return, it treats the string as a C null-terminated string. This string will be included in the SQLCA as a token for the error condition. At least the first part of this string will appear in the SQLCA or DB2 CLP message. However, the actual number of characters which will appear depends on the lengths of the other tokens, because DB2 may truncate the tokens to conform to the restrictive limit on total token length imposed by the SQLCA. Avoid using X'FF' in the text since this character is used to delimit tokens in the SQLCA.

The UDF code should not return more text than will fit in the VARCHAR(70) buffer which is passed to it. DB2 will attempt to determine if the UDF body has written beyond the end of this buffer by a few characters, SQLCODE -450 (SQLSTATE 39501). However, an overwrite by the UDF can cause unpredictable results or an abend, as it may not be detected by DB2.

DB2 assumes that any message tokens returned from the UDF to DB2 are in the same code page as the database. Your UDF should ensure that If this is the case. If you use the 7-bit invariant ASCII subset, your UDF can return the message tokens in any code page.

*scratchpad*

This argument is set by DB2 before calling the UDF. It is only present if the CREATE FUNCTION statement for the UDF specified the

SCRATCHPAD keyword. This argument is a structure, exactly like the structure used to pass a value of any of the LOB data types, with the following elements:

- An INTEGER containing the length of the scratchpad (always 100)
- The actual scratchpad (100 bytes in length), initialized to all binary 0's as follows. For scalar functions it is initialized before the first call, and not generally looked at or modified by DB2 thereafter.

For table functions, the scratchpad is initialized as above prior to the FIRST call to the UDF if FINAL CALL is specified on the CREATE FUNCTION. After this call, the scratchpad content is totally under control of the table function.

If NO FINAL CALL was specified or defaulted for a table function, then the scratchpad is initialized as above for each OPEN call, and the scratchpad content is completely under control of the table function between OPEN calls. (This can be very important for a table function used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then FINAL CALL must be specified in your CREATE FUNCTION statement. With FINAL CALL specified, in addition to the normal OPEN, FETCH and CLOSE calls, the table function will also receive FIRST and FINAL calls, for the purpose of scratchpad maintenance and resource release.)

The scratchpad can be mapped in your UDF using the same type as either a CLOB or a BLOB, since the argument passed has the same structure. See “How the SQL Data Types are Passed to a UDF” on page 303 for more information.

Ensure your UDF code does not make changes outside of the scratchpad buffer. DB2 attempts to determine if the UDF body has written beyond the end of this buffer by a few characters, SQLCODE -450 (SQLSTATE 39501), but a major overwrite by the UDF can cause unpredictable results, or an abend, and may not result in a graceful failure by DB2.

If a scalar UDF which uses a scratchpad is referenced in a subquery, DB2 may decide to refresh the scratchpad between invocations of the subquery. This refresh occurs after a *final-call* is made, if FINAL CALL is specified for the UDF.

#### *call-type*

This argument, if present, is set by DB2 before calling the UDF. For scalar functions this argument is only present if FINAL CALL is specified in the CREATE FUNCTION statement, but for table functions it is ALWAYS present. It follows the *scratchpad* argument; or

the *diagnostic-message* argument if the *scratchpad* argument is not present. This argument takes the form of an INTEGER value. Ensure that this argument definition in the UDF is appropriate for INTEGER. See “How the SQL Data Types are Passed to a UDF” on page 303 for more information.

Note that even though all the current possible values are listed below, your UDF should contain a switch or case statement which explicitly tests for all the expected values, rather than containing “if A do AA, else if B do BB, else it must be C so do CC” type logic. This is because it is possible that additional call types may be added in the future, and if you don’t explicitly test for condition C you will have trouble when new possibilities are added.

**Notes:**

1. For all the call-types, it may be appropriate for the UDF to set a *SQL-state* and *diagnostic-message* return value. This information will not be repeated in the following descriptions of each call-type. For all calls DB2 will take the indicated action as described previously for these arguments.
2. The include file `sqludf.h` is intended for use with UDFs and is described in “The UDF Include File: `sqludf.h`” on page 311. The file contains symbolic defines for the following call-type values, which are spelled out as constants.

For scalar functions *call-type* contains:

- 1 This is the *FIRST call* to the UDF for this statement. The scratchpad (if any) is set to binary zeros when the UDF is called. All argument values are passed, and the UDF should do whatever one-time initialization actions are required. In addition, a *FIRST* call to a scalar UDF is like a *NORMAL* call, in that it is expected to develop and return an answer.  
  
Note that if *SCRATCHPAD* is specified but *FINAL CALL* is not, then the UDF will not have this call-type argument to identify the very first call. Instead it will have to rely on the all-zero state of the scratchpad.
- 0 This is a *NORMAL call*. All the SQL input values are passed, and the UDF is expected to develop and return the result. The UDF may also return *SQL-state* and *diagnostic-message* information.
- 1 This is a *FINAL call*, that is no *SQL-argument* or *SQL-argument-ind* values are passed, and attempts to examine these values may cause unpredictable results. If a scratchpad

is also passed, it is untouched from the previous call. The UDF is expected to release resources at this point.

#### *Releasing resources.*

A scalar UDF is expected to release resources it has required, for example, memory. If FINAL CALL is specified for the UDF, then that FINAL call is a natural place to release resources, provided that SCRATCHPAD is also specified and is used to track the resource. If FINAL CALL is not specified, then any resource acquired should be released on the same call.

For table functions *call-type* contains:

- 2 This is the *FIRST call*, which only occurs if the FINAL CALL keyword was specified for the UDF. The scratchpad is set to binary zeros before this call. Argument values are passed to the table function, and it may choose to acquire memory or perform other one-time only resource initialization. Note that this is not an OPEN call, that call follows this one. On a FIRST call the table function should not return any data to DB2 as DB2 ignores the data.
- 1 This is the *OPEN call*. The scratchpad will be initialized if NO FINAL CALL is specified, but not necessarily otherwise. All SQL argument values are passed to the table function on OPEN. The table function should not return any data to DB2 on the OPEN call.
- 0 This is a *FETCH call*, and DB2 expects the table function to return either a row comprising the set of return values, or an end-of-table condition indicated by SQLSTATE value '02000'. If a scratchpad is passed to the UDF, then on entry it is untouched from the previous call.
- 1 This is a *CLOSE call* to the table function. It balances the OPEN call, and can be used to perform any external CLOSE processing (for example, closing a source file), and resource release (particularly for the NO FINAL CALL case).  
  
In cases involving a join or a subquery, the OPEN/FETCH.../CLOSE call sequences can repeat within the execution of a statement, but there is only one FIRST call and only one FINAL call. The FIRST and FINAL call only occur if FINAL CALL is specified for the table function.
- 2 This is a *FINAL call*, which only occurs if FINAL CALL was specified for the table function. It balances the FIRST call, and

occurs only once per execution of the statement. It is intended for the purpose of releasing resources.

#### *Releasing resources.*

Write UDFs to release any resources that they acquire. For table functions, there are two natural places for this release: the CLOSE call and the FINAL call. The CLOSE call balances each OPEN call and can occur multiple times in the execution of a statement. The FINAL call only occurs if FINAL CALL is specified for the UDF, and occurs only once per statement.

If you can apply a resource across all OPEN/FETCH/CLOSE sequences of the UDF, write the UDF to acquire the resource on the FIRST call and free it on the FINAL call. The scratchpad is a natural place to track this resource. For table functions, if FINAL CALL is specified, the scratchpad is initialized only before the FIRST call. If FINAL CALL is not specified, then it is reinitialized before each OPEN call.

If a resource is specific to each OPEN/FETCH/CLOSE sequence, write the UDF to free the resource on the CLOSE call. (Note that when a table function is in a subquery or join, it is very possible that there will be multiple occurrences of the OPEN/FETCH/CLOSE sequence, depending on how the DB2 Optimizer chooses to organize the execution of the statement.)

*dbinfo* This argument is set by DB2 before calling the UDF. It is only present if the CREATE FUNCTION statement for the UDF specifies the DBINFO keyword. The argument is the `sqludf_dbinfo` structure defined in the header file `sqludf.h`, which is discussed in “The UDF Include File: `sqludf.h`” on page 311. The variables in this structure that contain names and identifiers may be longer than the longest value possible in this release of DB2, but are defined this way for compatibility with future releases. You can use the length variable that complements each name and identifier variable to read or extract the portion of the variable that is actually used. The *dbinfo* structure contains the following elements:

1. Data base name length (`dbnamelen`)

The length of *data base name* below. This field is an unsigned short integer.

2. Data base name (`dbname`)

The name of the currently connected database. This field is a long identifier of 128 characters. The *data base name length* field

described above identifies the actual length of this field. It does not contain a null terminator or any padding.

3. Application Authorization ID Length (authidlen)

The length of *application authorization ID* below. This field is an unsigned short integer.

4. Application authorization ID (authid)

The application run time authorization ID. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *application authorization ID length* field described above identifies the actual length of this field.

5. Database code page (codepg)

This is a union of two 48-byte long structures; one is used by DB2 Universal Database, the other is reserved for future use. The structure used by DB2 Universal Database contains the following fields:

- a. SBCS. Single byte code page, an unsigned long integer.
- b. DBCS. Double byte code page, an unsigned long integer.
- c. COMP. Composite code page, an unsigned long integer.

6. Schema name length (tbschemalen)

The length of *schema name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

7. Schema name (tbschema)

Schema for the *table name* below. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *schema name length* field described above identifies the actual length of this field.

8. Table name length (tbnamelen)

The length of the *table name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.

9. Table name (tbnname)

This is the name of the table being updated or inserted. This field is set only if the UDF reference is the right-hand side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT statement. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *table name length* field described above, identifies the actual length of this field. The *schema name* field above, together with this field form the fully qualified table name.

10. Column name length (colnamelen)

Length of *column name* below. It contains a 0 (zero) if a column name is not passed. This field is an unsigned short integer.

11. Column name (colname)  
Under the exact same conditions as for table name, this field contains the name of the column being updated or inserted; otherwise not predictable. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *column name length* field described above, identifies the actual length of this field.
12. Version/Release number (ver\_rel)  
An 8 character field that identifies the product and its version, release, and modification level with the format *pppvrrm* where:
  - *ppp* identifies the product as follows:
    - DSN** DB2 for MVS/ESA or OS/390
    - ARI** SQL/DS
    - QSQ** DB2 Universal Database for AS/400
    - SQL** DB2 Universal Database
  - *vv* is a two digit version identifier.
  - *rr* is a two digit release identifier.
  - *m* is a one digit modification level identifier.
13. Platform (platform)  
The operating platform for the application server, as follows:
  - SQLUDF\_PLATFORM\_AIX** AIX
  - SQLUDF\_PLATFORM\_HP** HP-UX
  - SQLUDF\_PLATFORM\_MVS** OS/390
  - SQLUDF\_PLATFORM\_NT** Windows NT
  - SQLUDF\_PLATFORM\_OS2** OS/2
  - SQLUDF\_PLATFORM\_SUN** Solaris
  - SQLUDF\_PLATFORM\_WINDOWS**  
Windows 95 and Windows 98
  - SQLUDF\_PLATFORM\_UNKNOWN**  
Unknown platform

For additional platforms that are not contained in the above list, see the contents of the `sqludf.h` file.
14. Number of table function column list entries (numtfcol)  
The number of non-zero entries in the table function column list specified in the *table function column list* field below.
15. Reserved field (resd1)

This field is for future use. It is defined as 24 characters long.

16. Table function column list (tfcolumn)

If this is a table function, this field is a pointer to an array of short integers which is dynamically allocated by DB2. If this is a scalar function, this pointer is null.

This field is used only for table functions. Only the first *n* entries, where *n* is specified in the *number of table function column list entries* field, numtfcol, are of interest. *n* may be equal to 0, and is less than or equal to the number of result columns defined for the function in the RETURNS TABLE(...) clause of the CREATE FUNCTION statement. The values correspond to the ordinal numbers of the columns which this statement needs from the table function. A value of '1' means the first defined result column, '2' means the second defined result column, and so on, and the values may be in any order. Note that *n* could be equal to zero, that is, the variable numtfcol might be zero, for a statement similar to SELECT COUNT(\*) FROM TABLE(TF(...)) AS QQ, where no actual column values are needed by the query.

This array represents an opportunity for optimization. The UDF need not return all values for all the result columns of the table function, only those needed in the particular context, and these are the columns identified (by number) in the array. Since this optimization may complicate the UDF logic in order to gain the performance benefit, the UDF can choose to return every defined column.

17. Unique application identifier (appl\_id)

This field is a pointer to a C null-terminated string which uniquely identifies the application's connection to DB2. It is regenerated at each database connect.

The string has a maximum length of 32 characters, and its exact format depends on the type of connection established between the client and DB2. Generally it takes the form

<x>.<y>.<ts>

where the <x> and <y> vary by connection type, but the <ts> is a 12 character time stamp of the form YYMMDDHHMMSS, which is potentially adjusted by DB2 to ensure uniqueness.

Example: \*LOCAL.db2inst.980707130144

18. Reserved field (resd2)

This field is for future use. It is defined as 20 characters long.



## Summary of UDF Argument Use

The following is a summary of the arguments described above, and how you use them in the interface between DB2 and an external UDF.

For scalar functions, the arguments are:

- *SQL-argument*.  
This argument passes the values identified in the function reference from DB2 to the UDF. There is one of these arguments for each SQL argument.
- *SQL-result*.  
This argument passes the result value generated by the UDF back to DB2 and to the SQL statement where the function reference occurred.
- *SQL-argument-ind*.  
This argument corresponds positionally to *SQL-argument*, and tells the UDF whether or not a particular argument is null. There is one of these for each *SQL-argument*.
- *SQL-result-ind*.  
This argument is used by the UDF to report back to DB2 whether the function result in *SQL-result* contains nulls.
- *SQL-state* and *diagnostic-message*.  
These arguments are used by the UDF to signal exception information back to DB2.
- *function-name* and *specific-name*.  
These arguments are used by DB2 to pass the identity of the referenced function to the UDF.
- *scratchpad* and *call-type*.  
These arguments are used by DB2 to manage the saving of UDF state between calls. The *scratchpad* is created and initialized by DB2 and thereafter managed by the UDF. DB2 signals the type of call to the UDF using the *call-type* argument.
- *dbinfo*.  
A structure passed by DB2 to the UDF containing additional information.

A table function logically returns a table to the SQL statement that references it, but the physical interface between DB2 and the table function is row by row. For table functions, the arguments are:

- *SQL-argument*.  
This argument passes the values identified in the function reference from DB2 to the UDF. The argument has the same value for FETCH calls as it did for the OPEN and FIRST calls. There is one of these for each SQL argument.

- *SQL-result*.  
This argument is used to pass back the individual column values for the row being returned by the UDF. There is one of these arguments for each result column value defined in the RETURNS TABLE (...) clause of the CREATE FUNCTION statement.
- *SQL-argument-ind*.  
This argument corresponds positionally to *SQL-argument* values, and tells the UDF whether the particular argument is null. There is one of these for each SQL argument.
- *SQL-result-ind*.  
This argument is used by the UDF to report back to DB2 whether the individual column values returned in the table function output row is null. It corresponds positionally to the *SQL-result* argument.
- *SQL-state* and *diagnostic-message*.  
These arguments are used by the UDF to signal exception information and the end-of-table condition back to DB2.
- *function-name* and *specific-name*.  
These arguments are used by DB2 to pass the identity of the referenced function to the UDF.
- *scratchpad* and *call-type*.  
These arguments are used by DB2 to manage the saving of UDF state between calls. The *scratchpad* is created and initialized by DB2 and thereafter managed by the UDF. DB2 signals the type of call to the UDF using the *call-type* argument. For table functions these call types are OPEN, FETCH, CLOSE, and optionally FIRST and FINAL.
- *dbinfo*.  
This is a structure passed by DB2 to the UDF containing additional information.

Observe that the normal value outputs of the UDF, as well as the *SQL-result*, *SQL-result-ind*, and *SQL-state*, are returned to DB2 using arguments passed from DB2 to the UDF. Indeed, the UDF is written not to return anything in the functional sense (that is, the function's return type is void). See the void definition and the return statement in the following example:

```
#include ...
void SQL_API_FN divid(
 ... arguments ...)
{
 ... UDF body ...
 return;
}
```

In the above example, `SQL_API_FN` is a macro that specifies the calling convention for a function that may vary across supported operating systems. This macro is required when you write stored procedures or UDFs.

For programming examples of UDFs, see “Examples of UDF Code” on page 344.

## How the SQL Data Types are Passed to a UDF

This section identifies the valid types, for both UDF parameters and result, and specifies for each how the corresponding argument should be defined in your C or C++ language UDF. For type definitions in Java UDFs, see “Supported SQL Data Types” on page 535. Note that if you use the `sqludf.h` include file and the types defined there, you can automatically generate language variables and structures that are correct for the different data types and compilers. For example, for `BIGINT` you can use the `SQLUDF_BIGINT` data type to hide differences in the name of the 64 bit integer type between different compilers. This include file is discussed in “The UDF Include File: `sqludf.h`” on page 311.

*It is the data type for each function parameter defined in the CREATE FUNCTION statement that governs the format for argument values.* Promotions from the argument data type may be needed to get the value in that format. Such promotions are performed automatically by DB2 on the argument values; argument promotion is discussed in the *SQL Reference*.

For the function result, it is the data type specified in the `CAST FROM` clause of the `CREATE FUNCTION` statement that defines the format. If no `CAST FROM` clause is present, then the data type specified in the `RETURNS` clause defines the format.

In the following example, the presence of the `CAST FROM` clause means that the UDF body returns a `SMALLINT` and that DB2 casts the value to `INTEGER` before passing it along to the statement where the function reference occurs:

```
... RETURNS INTEGER CAST FROM SMALLINT ...
```

In this case the UDF must be written to generate a `SMALLINT`, as defined below. Note that the `CAST FROM` data type must be *castable* to the `RETURNS` data type, so one cannot just arbitrarily choose another data type. Casting between data types is discussed in the *SQL Reference*.

The following is a list of the SQL types and their C language representations. For a list of SQL type representations for Java, see “Supported SQL Data Types” on page 535. It includes information on whether each type is valid as a

parameter or a result. Also included are examples of how the types could appear as an argument definition in your C or C++ language UDF:

- **SMALLINT**

**Valid.** Represent in C as short.

When defining integer UDF parameters, consider using **INTEGER** rather than **SMALLINT** as DB2 does not promote **SMALLINT** arguments to **INTEGER**. For example, suppose you define a UDF as follows:

```
CREATE FUNCTION SIMPLE(SMALLINT)...
```

Example:

```
short *arg1; /* example for SMALLINT */
short *arg1_null_ind; /* example for any null indicator */
```

If you invoke the **SIMPLE** function using **INTEGER** data, (... **SIMPLE**(1)...), you will receive an **SQLCODE -440 (SQLSTATE 42884)** error indicating that the function was not found, and end-users of this function may not perceive the reason for the message. In the above example, 1 is an **INTEGER**, so you can either cast it to **SMALLINT** or define the parameter as **INTEGER**.

- **INTEGER or INT**

**Valid.** Represent in C as long.

Example:

```
long *arg2; /* example for INTEGER */
```

- **BIGINT**

**Valid.** Represent in C as **sqlint64**.

Example:

```
sqlint64 *arg3; /* example for INTEGER */
```

DB2 defines the **sqlint64** C language type to overcome differences between definitions of the 64 bit signed integer in compilers and operating systems. You must **#include sqludf.h** to pick up the definition.

- **DECIMAL(p,s) or NUMERIC(p,s)**

**Not valid**, because there is no C language representation. If you want to pass a decimal value, you must define the parameter to be of a data type castable from **DECIMAL** (for example **CHAR** or **DOUBLE**), and explicitly cast the argument to this type. In the case of **DOUBLE**, you do not need to explicitly cast a decimal argument to a **DOUBLE** parameter as DB2 promotes it automatically .

Suppose you have two columns, WAGE as DECIMAL(5,2) and HOURS as DECIMAL(4,1), and you wish to write a UDF to calculate weekly pay based on wage, number of hours worked and some other factors. The UDF could be as follows:

```
CREATE FUNCTION WEEKLY_PAY (DOUBLE, DOUBLE, ...)
 RETURNS DECIMAL(7,2) CAST FROM DOUBLE
 ...;
```

For the above UDF, the first two parameters correspond to the wage and number of hours. You invoke the UDF WEEKLY\_PAY in your SQL select statement as follows:

```
SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
```

Note that no explicit casting is required because the DECIMAL arguments are castable to DOUBLE.

Alternatively, you could define WEEKLY\_PAY with CHAR arguments as follows:

```
CREATE FUNCTION WEEKLY_PAY (VARCHAR(6), VARCHAR(5), ...)
 RETURNS DECIMAL (7,2) CAST FROM VARCHAR(10)
 ...;
```

You would invoke it as follows:

```
SELECT WEEKLY_PAY (CHAR(WAGE), CHAR(HOURS), ...) ...;
```

Observe the explicit casting that is required because DECIMAL arguments are not promotable to VARCHAR.

An advantage of using floating point parameters is that it is easier to perform arithmetic on the values in the UDF; an advantage of using character parameters is that it is always possible to exactly represent the decimal value. This is not always possible with floating point.

- REAL

**Valid.** Represent in C as float.

Example:

```
float *result; /* example for REAL */
```

- DOUBLE or DOUBLE PRECISION or FLOAT

**Valid.** Represent in C as double.

Example:

```
double *result; /* example for DOUBLE */
```

- CHAR(n) or CHARACTER(n) with or without the FOR BIT DATA modifier.

**Valid.** Represent in C as `char...[n+1]` (this is a C null-terminated string, the last character is a null, that is `X'00'`).

Example:

```
char arg1[14]; /* example for CHAR(13) */
char *arg1; /* also perfectly acceptable */
```

For a `CHAR(n)` parameter, DB2 always moves `n` bytes of data to the buffer and sets the `n+1` byte to null. For a `RETURNS CHAR(n)` value, DB2 always takes the `n` bytes and ignores the `n+1` byte. For this `RETURNS CHAR(n)` case, you are warned against the inadvertent inclusion of a null-character in the first `n` characters. DB2 will not recognize this as anything but a normal part of the data, and it might later on cause seemingly anomalous results if it was not intended.

If `FOR BIT DATA` is specified, exercise caution about using the normal C string handling functions in the UDF. Many of these functions look for a null to delimit the string, and the null-character (`X'00'`) could be a legitimate character in the middle of the data value.

When defining character UDF parameters, consider using `VARCHAR` rather than `CHAR` as DB2 does not promote `VARCHAR` arguments to `CHAR`. For example, suppose you define a UDF as follows:

```
CREATE FUNCTION SIMPLE(INT,CHAR(1))...
```

If you invoke the `SIMPLE` function using `VARCHAR` data, (`... SIMPLE(1, 'A') ...`), you will receive an `SQLCODE -440 (SQLSTATE 42884)` error indicating that the function was not found, and end-users of this function may not perceive the reason for the message. In the above example, `'A'` is `VARCHAR`, so you can either cast it to `CHAR` or define the parameter as `VARCHAR`.

- `VARCHAR(n)` `FOR BIT DATA` or `LONG VARCHAR` with or without the `FOR BIT DATA` modifier.

**Valid.** Represent in C as a structure similar to:

```
struct sqludf_vc_fbd
{
 unsigned short length; /* length of data */
 char data[1]; /* first char of data */
};
```

The `[1]` is merely to indicate an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

These values are not represented as C null-terminated strings because the null-character could legitimately be part of the data value. The length is explicitly passed to the UDF for parameters using the structure variable length. For the RETURNS clause, the length that is passed to the UDF is the length of the buffer. What the UDF body must pass back, using the structure variable length, is the actual length of the data value.

Example:

```
struct sqludf_vc_fbd *arg1; /* example for VARCHAR(n) FOR BIT DATA */
struct sqludf_vc_fbd *result; /* also for LONG VARCHAR FOR BIT DATA */
```

- VARCHAR(n) without FOR BIT DATA.

**Valid.** Represent in C as `char...[n+1]`. (This is a C null-terminated string.)

For a VARCHAR(n) parameter, DB2 will put a null in the (k+1) position, where k is the length of the particular occurrence. The C string-handling functions are thus well suited for manipulation of these values. For a RETURNS VARCHAR(n) value, the UDF body must delimit the actual value with a null, because DB2 will determine the result length from this null character.

Example:

```
char arg2[51]; /* example for VARCHAR(50) */
char *result; /* also perfectly acceptable */
```

- GRAPHIC(n)

**Valid.** Represent in C as `sqldbchar[n+1]`. (This is a null-terminated graphic string). Note that you can use `wchar_t[n+1]` on platforms where `wchar_t` is defined to be 2 bytes in length; however, `sqldbchar` is recommended. See “Selecting the `wchar_t` or `sqldbchar` Data Type” on page 520 for more information on these two data types.

For a GRAPHIC(n) parameter, DB2 moves n double-byte characters to the buffer and sets the following two bytes to null. Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the `WCHARTYPE NOCONVERT` precompiler option described in “The `WCHARTYPE` Precompiler Option” on page 521. For a RETURNS GRAPHIC(n) value, DB2 always takes the n double-byte characters and ignores the following bytes.

When defining graphic UDF parameters, consider using `VARGRAPHIC` rather than `GRAPHIC` as DB2 does not promote `VARGRAPHIC` arguments to `GRAPHIC`. For example, suppose you define a UDF as follows:

```
CREATE FUNCTION SIMPLE(GRAPHIC)...
```

If you invoke the `SIMPLE` function using `VARGRAPHIC` data, (`... SIMPLE('graphic_literal')...`), you will receive an `SQLCODE -440 (SQLSTATE 42884)` error indicating that the function was not found, and end-users of this function may not understand the reason for this message. In the above example, `graphic_literal` is a literal DBCS string that is

interpreted as VARGRAPHIC data, so you can either cast it to GRAPHIC or define the parameter as VARGRAPHIC.

Example:

```
sqldbchar arg1[14]; /* example for GRAPHIC(13) */
sqldbchar *arg1; /* also perfectly acceptable */
```

- **VARGRAPHIC(n)**

**Valid.** Represent in C as `sqldbchar[n+1]`. (This is a null-terminated graphic string). Note that you can use `wchar_t[n+1]` on platforms where `wchar_t` is defined to be 2 bytes in length; however, `sqldbchar` is recommended. See “Selecting the `wchar_t` or `sqldbchar` Data Type” on page 520 for more information on these two data types.

For a `VARGRAPHIC(n)` parameter, DB2 will put a graphic null in the (k+1) position, where k is the length of the particular occurrence. A graphic null refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the `WCHARTYPE NOCONVERT` precompiler option described in “The `WCHARTYPE` Precompiler Option” on page 521. For a `RETURNS VARGRAPHIC(n)` value, the UDF body must delimit the actual value with a graphic null, because DB2 will determine the result length from this graphic null character.

Example:

```
sqldbchar args[51], /* example for VARGRAPHIC(50) */
sqldbchar *result, /* also perfectly acceptable */
```

- **LONG VARGRAPHIC**

**Valid.** Represent in C as a structure:

```
struct sqludf_vg
{
 unsigned short length; /* length of data */
 sqldbchar data[1]; /* first char of data */
};
```

Note that in the above structure, you can use `wchar_t` in place of `sqldbchar` on platforms where `wchar_t` is defined to be 2 bytes in length, however, the use of `sqldbchar` is recommended. See “Selecting the `wchar_t` or `sqldbchar` Data Type” on page 520 for more information on these two data types.

The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed. Because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.



These are not represented as null-terminated graphic strings. The length, in double-byte characters, is explicitly passed to the UDF for parameters using the structure variable `length`. Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the `WCHARTYPE NOCONVERT` precompiler option described in “The `WCHARTYPE` Precompiler Option” on page 521. For the `RETURNS` clause, the length that is passed to the UDF is the length of the buffer. What the UDF body must pass back, using the structure variable `length`, is the actual length of the data value, in double byte characters.

Example:

```
struct sqludf_vg *arg1; /* example for VARGRAPHIC(n) */
struct sqludf_vg *result; /* also for LONG VARGRAPHIC */
```

- **DATE**

**Valid.** Represent in C same as `CHAR(10)`, that is as `char...[11]`. The date value is always passed to the UDF in ISO format: `yyyy-mm-dd`.

Example:

```
char arg1[11]; /* example for DATE */
char *result; /* also perfectly acceptable */
```

- **TIME**

**Valid.** Represent in C same as `CHAR(8)`, that is, as `char...[9]`. The time value is always passed to the UDF in ISO format: `hh.mm.ss`.

Example:

```
char *arg; /* example for DATE */
char result[9]; /* also perfectly acceptable */
```

- **TIMESTAMP**

**Valid.** Represent in C same as `CHAR(26)`, that is, as `char...[27]`. The timestamp value is always passed with format: `yyyy-mm-dd-hh.mm.ss.nnnnnn`.

Example:

```
char arg1[27]; /* example for TIMESTAMP */
char *result; /* also perfectly acceptable */
```

- **BLOB(n) and CLOB(n)**

**Valid.** Represent in C as a structure:

```
struct sqludf_lob
{
 unsigned long length; /* length in bytes */
 char data[1]; /* first byte of lob */
};
```

The [1] merely indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

These are not represented as C null-terminated strings. The length is explicitly passed to the UDF for parameters using the structure variable length. For the RETURNS clause, the length that is passed back to the UDF, is the length of the buffer. What the UDF body must pass back, using the structure variable length, is the actual length of the data value.

Example:

```
struct sqludf_lob *arg1; /* example for BLOB(n), CLOB(n) */
struct sqludf_lob *result;
```

- DBCLOB(n)

**Valid.** Represent in C as a structure:

```
struct sqludf_lob
{
 unsigned long length; /* length in graphic characters */
 sqldbchar data[1]; /* first byte of lob */
};
```

Note that in the above structure, you can use `wchar_t` in place of `sqldbchar` on platforms where `wchar_t` is defined to be 2 bytes in length, however, the use of `sqldbchar` is recommended. See “Selecting the `wchar_t` or `sqldbchar` Data Type” on page 520 for more information on these two data types.

The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed; because the address of the structure is passed, and not the actual structure, it just provides a way to use array logic.

These are not represented as null-terminated graphic strings. The length is explicitly passed to the UDF for parameters using the structure variable length. Data passed from DB2 to a UDF is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the `WCHARTYPE NOCONVERT` precompiler option described in “The `WCHARTYPE` Precompiler Option” on page 521. For the RETURNS clause, the length that is passed to the UDF is the length of the buffer. What the UDF body must pass back, using the structure variable length, is the actual length of the data value, with all of these lengths expressed in double byte characters.

Example:

```
struct sqludf_lob *arg1; /* example for DBCLOB(n) */
struct sqludf_lob *result;
```

- Distinct Types

**Valid or invalid depending on the base type.** Distinct types will be passed to the UDF in the format of the base type of the UDT, so may be specified if and only if the base type is valid.

Example:

```
struct sqludf_lob *arg1; /* for distinct type based on BLOB(n) */
double *arg2; /* for distinct type based on DOUBLE */
char res[5]; /* for distinct type based on CHAR(4) */
```

- Distinct Types AS LOCATOR, or any LOB type AS LOCATOR

The AS LOCATOR type modifier is valid only in UDF parameter and result definitions. It may only be used to modify the LOB types or any distinct type that is based on a LOB type. If you specify the type modifier, a four byte locator is passed to the UDF rather the entire LOB value.

Example:

```
sqludf_locator *arg1; /* locator argument */
sqludf_locator *result; /* locator result */
```

The type `udf_locator` is defined in the header file `sqludf.h`, which is discussed in “The UDF Include File: `sqludf.h`”. The use of these locators is discussed in “Using LOB Locators as UDF Parameters or Results” on page 335 .

## The UDF Include File: `sqludf.h`

This include file contains structures, definitions and values which are useful when writing your UDF. Its use is optional, however, and in the sample UDFs shown in “Examples of UDF Code” on page 344, some examples use the include file. When compiling your UDF, you need to reference the directory which contains this file. This directory is `sqllib/include`.

The `sqludf.h` include file is self-describing. Following is a brief summary of its content:

1. Structure definitions for the passed arguments which are structures:
  - VARCHAR FOR BIT DATA arguments and result
  - LONG VARCHAR (with or without FOR BIT DATA) arguments and result
  - LONG VARGRAPHIC arguments and result
  - All the LOB types, SQL arguments and result
  - The scratchpad
  - The `dbinfo` structure.

2. C language type definitions for all the SQL data types, for use in the definition of UDF arguments corresponding to SQL arguments and result having the data types. These are the definitions with names `SQLUDF_x` and `SQLUDF_x_FBD` where `x` is a SQL data type name, and `FBD` represents For Bit Data.  
Also included is a C language type for an argument or result which is defined with the `AS LOCATOR` appendage.
3. Definition of C language types for the *scratchpad* and *call-type* arguments, with an enum type definition of the *call-type* argument.
4. Macros for defining the standard *trailing* arguments, both with and without the inclusion of *scratchpad* and *call-type* arguments. This corresponds to the presence and absence of `SCRATCHPAD` and `FINAL CALL` keywords in the function definition. These are the *SQL-state*, *function-name*, *specific-name*, *diagnostic-message*, *scratchpad* and *call-type* UDF invocation arguments defined in “The Arguments Passed from DB2 to a UDF” on page 288. Also included are definitions for referencing these constructs, and the various valid `SQLSTATE` values.
5. Macros for testing whether the SQL arguments are null.
6. Function prototypes for the APIs which can be used to manipulate LOB values by means of LOB locators passed to the UDF.

Some of the UDF examples in the next section illustrate the inclusion and use of `sqludf.h`.

---

## Creating and Using Java User-Defined Functions

You can create and use UDFs in Java just as you would in other languages, with only a few minor differences. After you code the UDF, you register it with the database using the `CREATE FUNCTION` statement. Refer to the *SQL Reference* for information on registering a Java UDF using this statement. You can then refer to it in the SQL of your application. The UDF can be `FENCED` or `NOT FENCED`, and you can also use options to modify how the UDF is run. See “Changing How a Java UDF Runs” on page 314.

Some sample Java UDF method bodies are provided in the `UDFs.java` sample. You can find the associated `CREATE FUNCTION` statements and examples of calling those UDFs in the `CallUDFs.java` sample. See the `sqllib/samples/java` directory for the samples and `README` instructions for compiling and running the samples.

## Coding a Java UDF

In general, if you declare a UDF taking arguments of SQL types *t1*, *t2*, and *t3*, returning type *t4*, it will be called as a Java method with the expected Java signature:

```
public void name (T1 a, T2 b, T3 c, T4 d) {}
```

Where:

- *name* is the method name
- *T1* through *T4* are the Java types that correspond to SQL types *t1* through *t4*.
- *a*, *b*, and *c* are arbitrary variable names for the input arguments.
- *d* is an arbitrary variable name that represents the UDF result being computed.

For example, given a UDF called `sample!test3` that returns `INTEGER` and takes arguments of type `CHAR(5)`, `BLOB(10K)`, and `DATE`, DB2 expects the Java implementation of the UDF to have the following signature:

```
import COM.ibm.db2.app.*;
public class sample extends UDF {
 public void test3(String arg1, Blob arg2, String arg3,
 int result) { ... }
}
```

Java UDFs that implement table functions require more arguments. Beside the variables representing the input, an additional variable appears for each column in the resulting row. For example, a table function may be declared as:

```
public void test4(String arg1, int result1,
 Blob result2, String result3);
```

SQL NULL values are represented by Java variables that are not initialized. These variables have a value of zero if they are primitive types, and Java null if they are object types, in accordance with Java rules. To tell an SQL NULL apart from an ordinary zero, you can call the function `isNull` for any input argument:

```
{

 if (isNull(1)) { /* argument #1 was a SQL NULL */ }
 else { /* not NULL */ }
}
```

In the above example, the argument numbers start at one. The `isNull()` function, like the other functions that follow, are inherited from the `COM.ibm.db2.app.UDF` class.

To return a result from a scalar or table UDF, use the `set()` method in the UDF, as follows:

```

{
 set(2, value);
}

```

Where '2' is the index of an output argument, and value is a literal or variable of a compatible type. The argument number is the index in the argument list of the selected output. In the first example in this section, the `int result` variable has an index of 4; in the second, `result1` through `result3` have indices of 2 through 4. An output argument that is not set before the UDF returns will have a NULL value.

Like C modules used in UDFs and stored procedures, you cannot use the Java standard I/O streams (`System.in`, `System.out`, and `System.err`) in Java UDFs. For an example of a Java UDF, see the file `DB2Udf.java` in the `sqllib/samples/java` directory.

Remember that all Java class files that you use to implement a UDF must reside in the `sqllib/function` directory or an appropriate subdirectory. See "Where to Put Java Classes" on page 558.

## Changing How a Java UDF Runs

Typically, DB2 calls a UDF many times, once for each row of an input or result set in a query. If `SCRATCHPAD` is specified in the `CREATE FUNCTION` statement of the UDF, DB2 recognizes that some "continuity" is needed between successive invocations of the UDF, and therefore the implementing Java class is not instantiated for each call, but generally speaking once per UDF reference per statement. Generally it is instantiated before the first call and used thereafter, but may for table functions be instantiated more often. For more information, see the `NO FINAL CALL` execution model in the subsection which follows this one.

If, however, `NO SCRATCHPAD` is specified for a UDF, either a scalar or table function, then a clean instance is instantiated for each call to the UDF, by means of a call to the class constructor.

A scratchpad may be useful for saving information across calls to a UDF. While Java and OLE UDFs can either use instance variables or set the scratchpad to achieve continuity between calls, C and C++ UDFs must use the scratchpad. Java UDFs access the scratchpad with the `getScratchPad()` and `setScratchPad()` methods available in `COM.ibm.db2.app.UDF`.

For Java table functions that use a scratchpad, control when you get a new scratchpad instance by using the `FINAL CALL` or `NO FINAL CALL` option on the `CREATE FUNCTION` statement, as indicated by the execution models in "Table Function Execution Model for Java" on page 315.

The ability to achieve continuity between calls to a UDF by means of a scratchpad is controlled by the `SCRATCHPAD` and `NO SCRATCHPAD` option of `CREATE FUNCTION`, regardless of whether the DB2 scratchpad or instance variables are used.

For scalar functions, you use the same instance for the entire statement.

Please note that every reference to a Java UDF in a query is treated independently, even if the same UDF is referenced multiple times. This is the same as what happens for OLE, C and C++ UDFs as well. At the end of a query, if you specify the `FINAL CALL` option for a scalar function then the object's `close()` method is called. For table functions the `close()` method will always be invoked as indicated in the subsection which follows this one. If you do not define a `close()` method for your UDF class, then a stub function takes over and the event is ignored.

If you specify the `ALLOW PARALLEL` clause for a Java UDF in the `CREATE FUNCTION` statement, DB2 may elect to evaluate the UDF in parallel. If this occurs, several distinct Java objects may be created on different partitions. Each object receives a subset of the rows.

As with other UDFs, Java UDFs can be `FENCED` or `NOT FENCED`. `NOT FENCED` UDFs run inside the address space of the database engine; `FENCED` UDFs run in a separate process. Although Java UDFs cannot inadvertently corrupt the address space of their embedding process, they can terminate or slow down the process. Therefore, when you debug UDFs written in Java, you should run them as `FENCED` UDFs.

See “`COM.ibm.db2.app.UDF`” on page 663 for a description of the `COM.ibm.db2.app.UDF` interface. This interface describes other useful calls that you can make within a UDF, such as `setSQLstate` and `getDBInfo`.

## Table Function Execution Model for Java

For table functions written in Java, it is important to understand what happens at each point in DB2's processing of a given statement which is significant to the table function. The table which follows details this information. The bottom part of each box hints what the code might be written to do for a typical table function which pulls some information in from the Web. Covered are both the `NO FINAL CALL` and the `FINAL CALL` cases, assuming `SCRATCHPAD` in both cases.

| Point in scan time                                 | NO FINAL CALL<br>LANGUAGE JAVA<br>SCRATCHPAD                                                                                                                                                                                                                               | FINAL CALL<br>LANGUAGE JAVA<br>SCRATCHPAD                                                                                                                                                                                                                        |
|----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Before the first OPEN for the table function       | No calls.                                                                                                                                                                                                                                                                  | <ul style="list-style-type: none"> <li>Class constructor is called (means new scratchpad). UDF method is called with FIRST call.</li> <li>Constructor initializes class and scratchpad variables. Method connects to Web server.</li> </ul>                      |
| At each OPEN of the table function                 | <ul style="list-style-type: none"> <li>Class constructor is called (means new scratchpad). UDF method is called with OPEN call.</li> <li>Constructor initializes class and scratchpad variables. Method connect to Web server, and opens the scan for Web data.</li> </ul> | <ul style="list-style-type: none"> <li>UDF method is opened with OPEN call.</li> <li>Method opens the scan for whatever Web data it wants. (Might be able to avoid reopen after a CLOSE reposition, depending on what is saved in the scratchpad.)</li> </ul>    |
| At each FETCH for a new row of table function data | <ul style="list-style-type: none"> <li>UDF method is called with FETCH call.</li> <li>Method fetches and returns next row of data, or EOT.</li> </ul>                                                                                                                      | <ul style="list-style-type: none"> <li>UDF method is called with FETCH call.</li> <li>Method fetches and returns new row of data, or EOT.</li> </ul>                                                                                                             |
| At each CLOSE of the table function                | <ul style="list-style-type: none"> <li>UDF method is called with CLOSE call. <code>close()</code> method if it exists for class.</li> <li>Method closes its Web scan and disconnects from the Web server. <code>close()</code> does not need to do anything.</li> </ul>    | <ul style="list-style-type: none"> <li>UDF method is called with CLOSE call.</li> <li>Method might reposition to the top of the scan, or close the scan. It can save any state in the scratchpad, which will persist.</li> </ul>                                 |
| After the last CLOSE of the table function         | No calls.                                                                                                                                                                                                                                                                  | <ul style="list-style-type: none"> <li>UDF method is called with FINAL call. <code>close()</code> method is called if it exists for class.</li> <li>Method disconnects from the Web server. <code>close()</code> method does not need to do anything.</li> </ul> |



**Notes:**

1. By "UDF method" we mean the Java class method which implements the UDF. This is the method identified in the EXTERNAL NAME clause of the CREATE FUNCTION statement.
2. For table functions with NO SCRATCHPAD specified, the calls to the UDF method are as indicated in this table, but because the user is not asking for any continuity via a scratchpad, DB2 will cause a new object to be instantiated before each call, by calling the class constructor. It is not clear that table functions with NO SCRATCHPAD (and thus no continuity) can do very useful things, but they are supported.
3. These models are TOTALLY COMPATIBLE with what happens with the other UDF languages: C/C++ and OLE.

---

**Writing OLE Automation UDFs**

OLE (Object Linking and Embedding) automation is part of the OLE 2.0 architecture from Microsoft Corporation. With OLE automation, your applications, regardless of the language in which they are written, can expose their properties and methods in OLE automation objects. Other applications, such as Lotus Notes or Microsoft Exchange®, can then integrate these objects by taking advantage of these properties and methods through OLE automation.

The applications exposing the properties and methods are called OLE automation servers or objects, and the applications that access those properties and methods are called OLE automation controllers. OLE automation servers are COM components (objects) that implement the OLE IDispatch interface. An OLE automation controller is a COM client that communicates with the automation server through its IDispatch interface. COM (Component Object Model) is the foundation of OLE. For OLE automation UDFs, DB2 acts as an OLE automation controller. Through this mechanism, DB2 can invoke methods of OLE automation objects as external UDFs.

Note that this section assumes that you are familiar with OLE automation terms and concepts. This book does not present any introductory OLE material. For an overview of OLE automation, refer to *Microsoft Corporation: The Component Object Model Specification*, October 1995. For details on OLE automation, refer to *OLE Automation Programmer's Reference*, Microsoft Press, 1996, ISBN 1-55615-851-3.

For a list of sample applications included with the DB2 SDK that demonstrate OLE automation UDFs, see Table 37 on page 651.

## Creating and Registering OLE Automation UDFs

OLE automation UDFs are implemented as public methods of OLE automation objects. The OLE automation objects must be externally creatable by an OLE automation controller, in this case DB2, and support late binding (also called IDispatch-based binding). OLE automation objects must be registered in the Windows registration database (registry) with a class identifier (CLSID), and optionally, an OLE programmatic ID (progID) to identify the automation object. The progID can identify an in-process (.DLL) or local (.EXE) OLE automation server, or a remote server through DCOM (Distributed COM). OLE automation UDFs can be scalar functions or table functions.

After you code an OLE automation object, you need to register the methods of the object as UDFs using the SQL CREATE FUNCTION statement. Registering an OLE automation UDF is very similar to registering any external C or C++ UDF, but you must use the following options:

- LANGUAGE OLE
- FENCED, since OLE automation UDFs must run in FENCED mode

The external name consists of the OLE progID identifying the OLE automation object and the method name separated by ! (exclamation mark):

```
CREATE FUNCTION bcounter () RETURNS INTEGER
EXTERNAL NAME 'bert.bcounter!increment'
LANGUAGE OLE
FENCED
SCRATCHPAD
FINAL CALL
NOT DETERMINISTIC
NULL CALL
PARAMETER STYLE DB2SQL
NO SQL
NO EXTERNAL ACTION
DISALLOW PARALLEL;
```

The calling conventions for OLE method implementations are identical to the conventions for functions written in C or C++. An implementation of the above method in the BASIC language looks like the following (notice that in BASIC the parameters are by default defined as call by reference):

```
Public Sub increment(output As Long, _
 indicator As Integer, _
 sqlstate As String, _
 fname As String, _
 fspecname As String, _
 sqlmsg As String, _
 scratchpad() As Byte, _
 calltype As Long)
```

## Object Instance and Scratchpad Considerations

OLE automation UDFs (methods of OLE automation objects) are applied on instances of OLE automation objects. DB2 creates an object instance for each UDF reference in an SQL statement. An object instance can be reused for subsequent method invocations of the UDF reference in an SQL statement, or the instance can be released after the method invocation and a new instance is created for each subsequent method invocation. The proper behavior can be specified with the SCRATCHPAD option in the SQL CREATE FUNCTION statement. For the LANGUAGE OLE clause, the SCRATCHPAD option has the additional semantic compared to C or C++, that a single object instance is created and reused for the entire query, whereas if NO SCRATCHPAD is specified, a new object instance may be created each time a method is invoked. Separate instances are created for each UDF reference in an SQL statement.

Using the scratchpad allows a method to maintain state information in instance variables of the object, across function invocations. It also increases performance as an object instance is only created once and then reused for subsequent invocations.

### How the SQL Data Types are Passed to an OLE Automation UDF

DB2 handles the type conversions between SQL types and OLE automation types. The following table summarizes the supported data types and how they are mapped. The mapping of OLE automation types to data types of the implementing programming language, such as BASIC or C/C++, is described in Table 11 on page 321.

Table 10. Mapping of SQL and OLE Automation Datatypes

| SQL Type        | OLE Automation Type | OLE Automation Type Description                                         |
|-----------------|---------------------|-------------------------------------------------------------------------|
| SMALLINT        | short               | 16-bit signed integer                                                   |
| INTEGER         | long                | 32-bit signed integer                                                   |
| REAL            | float               | 32-bit IEEE floating-point number                                       |
| FLOAT or DOUBLE | double              | 64-bit IEEE floating-point number                                       |
| DATE            | DATE                | 64-bit floating-point fractional number of days since December 30, 1899 |
| TIME            | DATE                |                                                                         |
| TIMESTAMP       | DATE                |                                                                         |

Table 10. Mapping of SQL and OLE Automation Datatypes (continued)

| SQL Type                         | OLE Automation Type      | OLE Automation Type Description                                                                                                   |
|----------------------------------|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| CHAR( <i>n</i> )                 | BSTR                     | Length-prefixed string as described in the <i>OLE Automation Programmer's Reference</i> .                                         |
| VARCHAR( <i>n</i> )              | BSTR                     |                                                                                                                                   |
| LONG VARCHAR                     | BSTR                     |                                                                                                                                   |
| CLOB( <i>n</i> )                 | BSTR                     |                                                                                                                                   |
| GRAPHIC( <i>n</i> )              | BSTR                     | Length-prefixed string as described in the <i>OLE Automation Programmer's Reference</i> .                                         |
| VARGRAPHIC( <i>n</i> )           | BSTR                     |                                                                                                                                   |
| LONG GRAPHIC                     | BSTR                     |                                                                                                                                   |
| DBCLOB( <i>n</i> )               | BSTR                     |                                                                                                                                   |
| CHAR( <i>n</i> ) <sup>1</sup>    | SAFEARRAY[unsigned char] | 1-dim Byte() array of 8-bit unsigned data items. (SAFEARRAYs are described in the <i>OLE Automation Programmer's Reference</i> .) |
| VARCHAR( <i>n</i> ) <sup>1</sup> | SAFEARRAY[unsigned char] |                                                                                                                                   |
| LONG VARCHAR <sup>1</sup>        | SAFEARRAY[unsigned char] |                                                                                                                                   |
| BLOB( <i>n</i> )                 | SAFEARRAY[unsigned char] |                                                                                                                                   |

**Note:**

1. With FOR BIT DATA specified

Data passed between DB2 and OLE automation UDFs is passed as call by reference. SQL types such as BIGINT, DECIMAL, or LOCATORS, or OLE automation types such as Boolean or CURRENCY that are not listed in the table are not supported. Character and graphic data mapped to BSTR is converted from the database code page to the UCS-2 (also known as Unicode, IBM code page 13488) scheme. Upon return, the data is converted back to the database code page. These conversions occur regardless of the database code page. If code page conversion tables to convert from the database code page to UCS-2 and from UCS-2 to the database code page are not installed, you receive an SQLCODE -332 (SQLSTATE 57017).

## Implementing OLE Automation UDFs in BASIC and C++

You can implement OLE automation UDFs in any language. This section shows you how to implement OLE automation UDFs using BASIC or C++ as two sample languages.

Table 11 on page 321 shows the mapping of the various SQL data types to the intermediate OLE automation data types, and the data types in the language of interest (BASIC or C++). OLE data types are language independent, (that is, Table 10 on page 319 holds true for all languages).

Table 11. Mapping of SQL and OLE Data Types to BASIC and C++ Data Types

| SQL Type                                                                                                        | OLE Automation Type      | UDF Language |           |
|-----------------------------------------------------------------------------------------------------------------|--------------------------|--------------|-----------|
|                                                                                                                 |                          | BASIC Type   | C++ Type  |
| SMALLINT                                                                                                        | short                    | Integer      | short     |
| INTEGER                                                                                                         | long                     | Long         | long      |
| REAL                                                                                                            | float                    | Single       | float     |
| FLOAT or DOUBLE                                                                                                 | double                   | Double       | double    |
| DATE, TIME, TIMESTAMP                                                                                           | DATE                     | Date         | DATE      |
| CHAR( <i>n</i> ), VARCHAR( <i>n</i> ), LONG VARCHAR, CLOB( <i>n</i> )                                           | BSTR                     | String       | BSTR      |
| GRAPHIC( <i>n</i> ), VARGRAPHIC( <i>n</i> ), LONG GRAPHIC, DBCLOB( <i>n</i> )                                   | BSTR                     | String       | BSTR      |
| CHAR( <i>n</i> ) <sup>1</sup> , VARCHAR( <i>n</i> ) <sup>1</sup> , LONG VARCHAR <sup>1</sup> , BLOB( <i>n</i> ) | SAFEARRAY[unsigned char] | Byte()       | SAFEARRAY |

**Note:**

1. With FOR BIT DATA specified

**OLE Automation UDFs in BASIC**

To implement OLE automation UDFs in BASIC you need to use the BASIC data types corresponding to the SQL data types mapped to OLE automation types.

The BASIC declaration of the bcounter OLE automation UDF in “Creating and Registering OLE Automation UDFs” on page 318 looks like the following:

```
Public Sub increment(output As Long, _
 indicator As Integer, _
 sqlstate As String, _
 fname As String, _
 fspecname As String, _
 sqlmsg As String, _
 scratchpad() As Byte, _
 calltype As Long)
```

You can find an example of an OLE table automation in “Example: Mail OLE Automation Table Function in BASIC” on page 369.

**OLE Automation UDFs in C++**

Table 11 shows the C++ data types that correspond to the SQL data types and how they map to OLE automation types.

The C++ declaration of the increment OLE automation UDF is as follows:

```
STDMETHODIMP Ccounter::increment (long *output,
 short *indicator,
 BSTR *sqlstate,
 BSTR *fname,
 BSTR *fspecname,
 BSTR *sqlmsg,
 SAFEARRAY **scratchpad,
 long *calltype);
```

OLE supports type libraries that describe the properties and methods of OLE automation objects. Exposed objects, properties, and methods are described in the Object Description Language (ODL). The ODL description of the above C++ method is as follows:

```
HRESULT increment ([out] long *output,
 [out] short *indicator,
 [out] BSTR *sqlstate,
 [in] BSTR *fname,
 [in] BSTR *fspecname,
 [out] BSTR *sqlmsg,
 [in,out] SAFEARRAY (unsigned char) *scratchpad,
 [in] long *calltype);
```

The ODL description allows the specification whether a parameter is an input (in), output (out) or input/output (in,out) parameter. For an OLE automation UDF, the UDF input parameters and its input indicators are specified as [in] parameters, and UDF output parameters and its output indicators as [out] parameters. For the UDF trailing arguments, sqlstate is an [out] parameter, function name and function specific name are [in] parameters, scratchpad is an [in,out] parameter, and call type is an [in] parameter.

Scalar functions contain one output parameter and output indicator, whereas table functions contain multiple output parameters and output indicators corresponding to the RETURN columns of the CREATE FUNCTION statement.

OLE automation defines the BSTR data type to handle strings. BSTR is defined as a pointer to OLECHAR: typedef OLECHAR \*BSTR. For allocating and freeing BSTRs, OLE imposes the rule, that the callee frees a BSTR passed in as a by-reference parameter before assigning the parameter a new value. This rule means the following for DB2 and OLE automation UDFs. The same rule applies for one-dimensional byte arrays which are received by the callee as SAFEARRAY\*\*:

- [in] parameters: DB2 allocates and frees [in] parameters.
- [out] parameters: DB2 passes in a pointer to NULL. The [out] parameter must be allocated by the callee and is freed by DB2.

- [in,out] parameters: DB2 initially allocates [in,out] parameters. They can be freed and re-allocated by the callee. As is true for [out] parameters, DB2 frees the final returned parameter.

All other parameters are passed as pointers. DB2 allocates and manages the referenced memory.

OLE automation provides a set of data manipulation functions for dealing with BSTRs and SAFEARRAYs. The data manipulation functions are described in the *OLE Automation Programmer's Reference*.

The following C++ UDF returns the first 5 characters of a CLOB input parameter:

```
// UDF DDL: CREATE FUNCTION crunch (clob(5k)) RETURNS char(5)

STDMETHODIMP Cobj::crunch (BSTR *in, // CLOB(5K)
 BSTR *out, // CHAR(5)
 short *indicator1, // input indicator
 short *indicator2, // output indicator
 BSTR *sqlstate, // pointer to NULL
 BSTR *fname, // pointer to function name
 BSTR *fspecname, // pointer to specific name
 BSTR *msgtext) // pointer to NULL
{
 // Allocate BSTR of 5 characters
 // and copy 5 characters of input parameter

 // out is an [out] parameter of type BSTR, that is,
 // it is a pointer to NULL and the memory does not have to be freed.
 // DB2 will free the allocated BSTR.

 *out = SysAllocStringLen (*in, 5);
 return NOERROR;
};
```

An OLE automation server can be implemented as *creatable single-use* or *creatable multi-use*. With *creatable single-use*, each client (that is, a DB2 FENCED process) connecting with CoGetObject to an OLE automation object will use its own instance of a class factory, and run a new copy of the OLE automation server if necessary. With *creatable multi-use*, many clients connect to the same class factory. That is, each instantiation of a class factory is supplied by an already running copy of the OLE server, if any. If there are no copies of the OLE server running, a copy is automatically started to supply the class object. The choice between single-use and multi-use OLE automation servers is yours, when you implement your automation server. A single-use server is recommended for better performance.

---

## OLE DB Table Functions

Microsoft OLE DB is a set of OLE/COM interfaces that provide applications with uniform access to data stored in diverse information sources. The OLE DB component DBMS architecture defines OLE DB consumers and OLE DB providers. An OLE DB consumer is any system or application that consumes OLE DB interfaces; an OLE DB provider is a component that exposes OLE DB interfaces. There are two classes of OLE DB providers: *OLE DB data providers*, which own data and expose their data in tabular format as a rowset; and *OLE DB service providers*, which do not own their own data, but encapsulate some services by producing and consuming data through OLE DB interfaces.

DB2 Universal Database simplifies the creation of OLE DB applications by enabling you to define table functions that access an OLE DB data source. DB2 is an OLE DB consumer that can access any OLE DB data or service provider. You can perform operations including GROUP BY, JOIN, and UNION on data sources that expose their data through OLE DB interfaces. For example, you can define an OLE DB table function to return a table from a Microsoft Access database or a Microsoft Exchange address book, then create a report that seamlessly combines data from this OLE DB table function with data in your DB2 database.

Using OLE DB table functions reduces your application development effort by providing built-in access to any OLE DB provider. For C, Java, and OLE automation table functions, the developer needs to implement the table function, whereas in the case of OLE DB table functions, a generic built-in OLE DB consumer interfaces with any OLE DB provider to retrieve data. You only need to register a table function of language type OLEDB, and refer to the OLE DB provider and the relevant rowset as a data source. You do not have to do any UDF programming to take advantage of OLE DB table functions.

To use OLE DB table functions with DB2 Universal Database, you must install OLE DB 2.0 or later, available from Microsoft at <http://www.microsoft.com>. If you attempt to invoke an OLE DB table function without first installing OLE DB, DB2 issues SQLCODE 465, SQLSTATE 58032, reason code 35. For the system requirements and OLE DB providers available for your data sources, refer to your data source documentation. For a list of samples that define and use OLE DB table functions, see “Appendix B. Sample Programs” on page 633. For the OLE DB specification, see the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.



## Creating an OLE DB Table Function

To define an OLE DB table function with a single CREATE FUNCTION statement, you must:

- define the table that the OLE DB provider returns
- specify LANGUAGE OLEDB
- identify the OLE DB rowset and provide an OLE DB provider connection string in the EXTERNAL NAME clause

OLE DB data sources expose their data in tabular form, called *rowset*. A rowset is a set of rows, each having a set of columns. The RETURNS TABLE clause includes only the columns relevant to the user. The binding of table function columns to columns of a rowset at an OLE DB data source is based on column names. If the OLE DB provider is case sensitive, place the column names in quotation marks; for example, "UPPERcase". For information on rowset names which can be fully qualified names, see "Fully Qualified Rowset Names" on page 327. For the mapping of OLE DB data types to DB2 data types, see "Supported OLE DB Data Types" on page 329. For the complete syntax of the CREATE FUNCTION statement and the rules for the EXTERNAL NAME clause, refer to the *SQL Reference*.

The EXTERNAL NAME clause can take either of the following forms:

```
'server!rowset'
or
'!rowset!connectstring'
```

where:

**server** identifies a server registered with CREATE SERVER statement

**rowset**

identifies a rowset, or table, exposed by the OLE DB provider; this value should be empty if the table has an input parameter to pass through command text to the OLE DB provider.

**connectstring**

contains initialization properties needed to connect to an OLE DB provider. For the complete syntax and semantics of the connection string, see the "Data Link API of the OLE DB Core Components" in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

You can use a *connection string* in the EXTERNAL NAME clause of a CREATE FUNCTION statement, or specify the *CONNECTSTRING* option in a CREATE SERVER statement.

For example, you can define an OLE DB table function and return a table from a Microsoft Access database with the following CREATE FUNCTION and SELECT statements:

```
CREATE FUNCTION orders ()
 RETURNS TABLE (orderid INTEGER, ...)
 LANGUAGE OLEDB
 EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
 Data Source=c:\msdasdk\bin\oledb\nwind.mdb';

SELECT orderid, DATE(orderdate) AS orderdate,
 DATE(shippeddate) AS shippeddate
FROM TABLE(orders()) AS t
WHERE orderid = 10248;
```

Instead of putting the connection string in the EXTERNAL NAME clause, you can create and use a server name. For example, assuming you have defined the server Nwind as described in “Defining a Server Name for an OLE DB Provider” on page 328, you could use the following CREATE FUNCTION statement:

```
CREATE FUNCTION orders ()
 RETURNS TABLE (orderid INTEGER, ...)
 LANGUAGE OLEDB
 EXTERNAL NAME 'Nwind!orders';
```

OLE DB table functions also allow you to specify one input parameter of any character string data type. Use the input parameter to pass command text directly to the OLE DB provider. If you define an input parameter, do not provide a rowset name in the EXTERNAL NAME clause. DB2 passes the command text to the OLE DB provider for execution and the OLE DB provider returns a rowset to DB2. Column names and data types of the resulting rowset need to be compatible with the RETURNS TABLE definition in the CREATE FUNCTION statement. Since binding to the column names of the rowset is based on matching column names, you must ensure that you name the columns properly.

The following example registers an OLE DB table function, which retrieves store information from a Microsoft SQL Server 7.0™ database. The connection string is provided in the EXTERNAL NAME clause. Since the table function has an input parameter to pass through command text to the OLE DB provider, the rowset name is not specified in the EXTERNAL NAME clause. The query example passes in a SQL command text which retrieves information about the top three stores from a SQL Server database.

```
CREATE FUNCTION favorites (varchar(600))
 RETURNS TABLE (store_id char (4), name varchar (41), sales integer)
 SPECIFIC favorites
 LANGUAGE OLEDB
 EXTERNAL NAME '!!!Provider=SQLOLEDB.1;Persist Security Info=False;
 User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
```

```

Locale Identifier=1033;Use Procedure for Prepare=1;
Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
OLE DB Services=CLIENTCURSOR;';

```

```

SELECT *
FROM TABLE (favorites (' select top 3 sales.stor_id as store_id, ' |
 ' stores.stor_name as name, ' |
 ' sum(sales.qty) as sales ' |
 ' from sales, stores ' |
 ' where sales.stor_id = stores.stor_id ' |
 ' group by sales.stor_id, stores.stor_name ' |
 ' order by sum(sales.qty) desc')) as f;

```

## Fully Qualified Rowset Names

Some rowsets need to be identified in the EXTERNAL NAME clause through a *fully qualified name*. A fully qualified name incorporates either or both of the following:

- the associated catalog name, which requires the following information:
  - whether the provider supports catalog names
  - where to put the catalog name in the fully qualified name
  - which catalog name separator to use
- the associated schema name, which requires the following information:
  - whether the provider supports schema names
  - which schema name separator to use

For information on the support offered by your OLE DB provider for catalog and schema names, refer to the documentation of the literal information of your OLE DB provider.

If DBLITERAL\_CATALOG\_NAME is not NULL in the literal information of your provider, use a catalog name and the value of DBLITERAL\_CATALOG\_SEPARATOR as a separator. To determine whether the catalog name goes at the beginning or the end of the fully qualified name, refer to the value of DBPROP\_CATALOGLOCATION in the property set DBPROPSET\_DATASOURCEINFO of your OLE DB provider.

If DBLITERAL\_SCHEMA\_NAME is not NULL in the literal information of your provider, use a schema name and the value of DBLITERAL\_SCHEMA\_SEPARATOR as a separator.

If the names contain special characters or match keywords, enclose the names in the quote characters specified for your OLE DB provider. The quote characters are defined in the literal information of your OLE DB provider as DBLITERAL\_QUOTE\_PREFIX and DBLITERAL\_QUOTE\_SUFFIX. For example, in the

following EXTERNAL NAME the specified rowset includes catalog name *pubs* and schema name *dbo* for a rowset called *authors*, with the quote character " used to enclose the names.

```
EXTERNAL NAME '!"pubs"."dbo"."authors"!Provider=SQLOLEDB.1;...';
```

For more information on constructing fully qualified names, refer to *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998, and the documentation for your OLE DB provider.

## Defining a Server Name for an OLE DB Provider

To provide a server name for an OLE DB data source that you can use for many CREATE FUNCTION statements, use the CREATE SERVER statement as follows:

- provide a name that identifies the OLE DB provider within DB2
- specify WRAPPER OLEDB
- provide connection information in the CONNECTSTRING option

For example, you can define the server name Nwind for the Microsoft Access OLE DB provider with the following CREATE SERVER statement:

```
CREATE SERVER Nwind
 WRAPPER OLEDB
 OPTIONS (CONNECTSTRING 'Provider=Microsoft.Jet.OLEDB.3.51;
 Data Source=c:\msdasdk\bin\oledb\nwind.mdb');
```

You can then use the server name Nwind to identify the OLE DB provider in a CREATE FUNCTION statement, for example:

```
CREATE FUNCTION orders ()
 RETURNS TABLE (orderid INTEGER, ...)
 LANGUAGE OLEDB
 EXTERNAL NAME 'Nwind!orders';
```

For the complete syntax of the CREATE SERVER statement, refer to the *SQL Reference*. For information on user mappings for OLE DB providers, see “Defining a User Mapping”.

## Defining a User Mapping

You can provide *user mappings* for your DB2 users to provide access to OLE DB data sources with an alternate username and password. To map usernames for specific users, you can define user mappings with the CREATE USER MAPPING statement. To provide a user mapping shared by all users, add the username and password to the connection string of your CREATE FUNCTION or CREATE SERVER statement. For example, to create a specific user mapping for the DB2 user JOHN on the OLE DB server Nwind, use the following CREATE USER MAPPING statement:

```
CREATE USER MAPPING FOR john
 SERVER Nwind
 OPTIONS (REMOTE_AUTHID 'dave', REMOTE_PASSWORD 'mypwd');
```

To provide the equivalent access to all of the DB2 users that call the OLE DB table function orders, use the following CONNECTSTRING either in a CREATE FUNCTION or CREATE SERVER statement:

```
CREATE FUNCTION orders ()
 RETURNS TABLE (orderid INTEGER, ...)
 LANGUAGE OLEDB
 EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;User ID=dave;
 Password=mypwd;Data Source=c:\msdasdk\bin\oledb\nwind.mdb';
```

DB2 applies the following user mapping rules:

- If a user mapping is defined, DB2 uses the mapped authorization information to connect to the OLE DB provider, potentially overwriting an existing userid and password in the CONNECTSTRING.
- If no user mapping is defined, DB2 uses the authorization information from the CONNECTSTRING, if the information is provided.
- If no user mapping is defined and no authorization information is provided in the CONNECTSTRING, DB2 uses the current DB2 authorization information if the provider supports authorization.

For the complete syntax of the CREATE USER MAPPING statement, refer to the *SQL Reference*.

## Supported OLE DB Data Types

The following table shows how DB2 data types map to the OLE DB data types described in *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998. Use the mapping table to define the appropriate RETURNS TABLE columns in your OLE DB table functions. For example, if you define an OLE DB table function with a column of data type INTEGER, DB2 requests the data from the OLE DB provider as DBTYPE\_I4.

For mappings of OLE DB provider source data types to OLE DB data types, refer to the OLE DB provider documentation. For examples of how the ANSI SQL, Microsoft Access, and Microsoft SQL Server providers might map their respective data types to OLE DB data types, refer to the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

Table 12. Mapping DB2 Data Types to OLE DB

| DB2 Data Type             | OLE DB Data Type      |
|---------------------------|-----------------------|
| SMALLINT                  | DBTYPE_I2             |
| INTEGER                   | DBTYPE_I4             |
| BIGINT                    | DBTYPE_I8             |
| REAL                      | DBTYPE_R4             |
| FLOAT/DOUBLE              | DBTYPE_R8             |
| DEC (p, s)                | DBTYPE_NUMERIC (p, s) |
| DATE                      | DBTYPE_DBDATE         |
| TIME                      | DBTYPE_DBTIME         |
| TIMESTAMP                 | DBTYPE_DBTIMESTAMP    |
| CHAR(N)                   | DBTYPE_STR            |
| VARCHAR(N)                | DBTYPE_STR            |
| LONG VARCHAR              | DBTYPE_STR            |
| CLOB(N)                   | DBTYPE_STR            |
| CHAR(N) FOR BIT DATA      | DBTYPE_BYTES          |
| VARCHAR(N) FOR BIT DATA   | DBTYPE_BYTES          |
| LONG VARCHAR FOR BIT DATA | DBTYPE_BYTES          |
| BLOB(N)                   | DBTYPE_BYTES          |
| GRAPHIC(N)                | DBTYPE_WSTR           |
| VARGRAPHIC(N)             | DBTYPE_WSTR           |
| LONG GRAPHIC              | DBTYPE_WSTR           |
| DBCLOB(N)                 | DBTYPE_WSTR           |

**Note:** OLE DB data type conversion rules are defined in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998.

For example:

- To retrieve the OLE DB data type DBTYPE\_CY, the data may get converted to OLE DB data type DBTYPE\_NUMERIC(19,4) which maps to DB2 data type DEC(19,4).
- To retrieve the OLE DB data type DBTYPE\_I1, the data may get converted to OLE DB data type DBTYPE\_I2 which maps to DB2 data type SMALLINT.
- To retrieve the OLE DB data type DBTYPE\_GUID, the data may get converted to OLE DB data type DBTYPE\_BYTES which maps to DB2 data type CHAR(12) FOR BIT DATA.

---

## Scratchpad Considerations

The factors influencing whether your UDF should use a scratchpad or not are important enough to warrant this special section. Other coding considerations are discussed in “Other Coding Considerations” on page 340.

**It is important that you code UDFs to be re-entrant.** This is primarily due to the fact that many references to the UDF may use the same copy of the function body. In fact, these *many references* may even be in different statements or applications. However, note that functions may need or want to save state from one invocation to the next. Two categories of these functions are:

1. Functions that, to be correct, depend on saving state.

An example of such a function is a simple *counter* function which returns a '1' the first time it is called, and increments the result by one each successive call. Such a function could be used to number the rows of a SELECT result:

```
SELECT counter(), a, b+c, ...
FROM tablex
WHERE ...
```

This type of function is NOT DETERMINISTIC (or VARIANT). Its output does not depend solely on the values of its SQL arguments. This *counter* function is shown in “Example: Counter” on page 353.

2. Functions where the performance can be improved by the ability to perform some initialization actions one time only.

An example of such a function, which may be a part of a document application, is a *match* function, which returns 'Y' if a given document contains a given string, and 'N' otherwise:

```
SELECT docid, doctitle, docauthor
FROM docs
WHERE match('myocardial infarction', docid) = 'Y'
```

This statement returns all the documents containing the particular text string value represented by the first argument. What *match* would like to do is:

- First time only.

Retrieve a list of all the document IDs which contain the string *myocardial infarction* from the document application which is maintained outside of DB2. This retrieval is a costly process, so the function would like to do it only one time, and save the list somewhere handy for subsequent calls.

- On each call.

Use the list of document IDs saved during this first call to see if the document ID which is passed as the second argument is contained in the list.

This particular *match* function is DETERMINISTIC (or NOT VARIANT). Its answer only depends on its input argument values. What is shown here is a function whose performance, not correctness, depends on the ability to save information from one call to the next.

Both of these needs are met by the ability to specify a SCRATCHPAD in the CREATE FUNCTION statement:

```
CREATE FUNCTION counter()
 RETURNS int ... SCRATCHPAD;
CREATE FUNCTION match(varchar(200), char(15))
 RETURNS char(1) ... SCRATCHPAD;
```

This SCRATCHPAD keyword tells DB2 to allocate and maintain a scratchpad for the function. DB2 initializes the scratchpad to binary zeros. If the table function is specified with NO FINAL CALL (the default), DB2 refreshes the scratchpad before each OPEN call. If you specify the table function option FINAL CALL, DB2 does not examine or change the content of the scratchpad thereafter. The scratchpad is passed to the function on each invocation. The function can be re-entrant, and DB2 preserves its state information in the scratchpad.

So for the *counter* example, the last value returned could be kept in the scratchpad. And the *match* example could keep the list of documents in the scratchpad if the scratchpad is big enough, or otherwise could allocate memory for the list and keep the address of the acquired memory in the scratchpad.

Because it is recognized that a UDF may want to acquire system resources, the UDF can be defined with the FINAL CALL keyword. This keyword tells DB2 to call the UDF at end-of-statement processing so that the UDF can release its system resources. In particular, since the scratchpad is of fixed size, the UDF may want to allocate memory for itself and thus uses the final call to free the memory. For example the *match* function above cannot predict how many documents will match the given text string. So a better definition for *match* is:

```
CREATE FUNCTION match(varchar(200), char(15))
 RETURNS char(1) ... SCRATCHPAD FINAL CALL;
```

Note that for UDFs that use a scratchpad and are referenced in a subquery, DB2 may decide to make a final call (if the UDF is so specified) and refresh the scratchpad between invocations of the subquery. You can protect yourself



against this possibility, if your UDFs are ever used in subqueries, by defining the UDF with FINAL CALL and using the call-type argument, or by always checking for the *binary zero* condition.

If you do specify FINAL CALL, please note that your UDF receives a call of type FIRST. This could be used to acquire and initialize some persistent resource.

---

## Table Function Considerations

A table function is an external UDF which delivers a table to the SQL in which it is referenced. A table function reference is only valid in a FROM clause of a SELECT. When using table functions, observe the following:

- Even though a table function delivers a table, the physical interface between DB2 and the UDF is one-row-at-a-time. There are five types of call made to a table function: OPEN, FETCH, CLOSE, FIRST, and FINAL. The existence of FIRST and FINAL calls depends on how you define the UDF. The same *call-type* mechanism that can be used for scalar functions is used to distinguish these calls.
- The standard interface used between DB2 and user-defined scalar functions is extended to accommodate table functions. The *SQL-result* argument repeats for table functions, each instance corresponding to a column to be returned as defined in the RETURNS TABLE clause of the CREATE FUNCTION statement. The *SQL-result-ind* argument likewise repeats, each instance related to the corresponding *SQL-result* instance.
- Not every result column defined in the RETURNS clause of the CREATE FUNCTION statement for the table function has to be returned. The DBINFO keyword of CREATE FUNCTION, and corresponding *dbinfo* argument enable the optimization that only those columns needed for a particular table function reference need be returned.
- The individual column values returned conform in format to the values returned by scalar functions.
- The CREATE FUNCTION statement for a table function has a *CARDINALITY n* specification. This specification enables the definer to inform the DB2 optimizer of the approximate size of the result so that the optimizer can make better decisions when the function is referenced.

Regardless of what has been specified as the *CARDINALITY* of a table function, exercise caution against writing a function with infinite cardinality, that is, a function that always returns a row on a FETCH call. There are many situations where DB2 expects the *end-of-table* condition, as a catalyst within its query processing. Using GROUP BY or ORDER BY are examples where this is the case. DB2 cannot form the groups for aggregation until *end-of-table* is reached, and it cannot sort until it has all the data. So a table

function that never returns the end-of-table condition (SQL-state value '02000') can cause an infinite processing loop if you use it with a GROUP BY or ORDER BY clause.

---

## Table Function Error Processing

The error processing model for table function calls is as follows:

1. If FIRST call fails, no further calls are made.
2. If FIRST call succeeds, the nested OPEN, FETCH, and CLOSE calls are made, and the FINAL call is always made.
3. If OPEN call fails, no FETCH or CLOSE call is made.
4. If OPEN call succeeds, then FETCH and CLOSE calls are made.
5. If a FETCH call fails, no further FETCH calls are made, but the CLOSE call is made.

**Note:** This model describes the ordinary error processing for scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model may not be made. For example, for a FENCED UDF, if the db2udf fenced process is somehow prematurely terminated, DB2 cannot make the indicated calls.

---

## Scalar Function Error Processing

The error processing model for scalar UDFs which are defined with the FINAL CALL specification is as follows:

- If FIRST call fails, no further calls are made.
- If FIRST call succeeds, then further NORMAL calls are made as warranted by the processing of the statement, and a FINAL call is always made.
- If NORMAL call fails, no further NORMAL calls are made, but the FINAL call is made (if you have specified FINAL CALL).

This means that if an error is returned on a FIRST call, the UDF must clean up before returning, because no FINAL call will be made.

The error processing model for table functions is defined in “Table Function Considerations” on page 333 section of this chapter.

**Note:** This model describes the ordinary error processing for scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model may not be made. For example, for a FENCED UDF, if the db2udf fenced process is somehow prematurely terminated, DB2 cannot make the indicated calls.

---

## Using LOB Locators as UDF Parameters or Results

You can append AS LOCATOR to any of the LOB data types, or any distinct types based on LOB types in a CREATE FUNCTION statement. This applies both to the parameters that are passed and the results that are returned. When this happens, DB2 does the following:

- For parameters, DB2 passes a four byte locator instead of the entire LOB value. This locator can be used in a number of ways, involving a set of special APIs (described below), to retrieve and manipulate the actual bytes. The savings are clear for the case where the UDF only needs a few bytes of the value:

- |                    |                                                                                                        |
|--------------------|--------------------------------------------------------------------------------------------------------|
| <b>Storage</b>     | Memory for the entire LOB need not be allocated.                                                       |
| <b>Performance</b> | Materializing the entire value can take a significant amount of I/O time and byte moving instructions. |
- For results, returns a four-byte locator instead of the entire LOB value. Again there can be storage and performance benefits.

Do not modify the locator values as this makes them unusable, and the APIs will return errors.

These special APIs can only be used in UDFs which are defined as NOT FENCED. This implies that these UDFs in test phase should not be used on a production database, because of the possibility that a UDF with bugs could cause the system harm. When operating on a test database, no lasting harm can result from the UDF if it should have bugs. When the UDF is known to be free of errors it can then be applied to the production database.

The APIs which follow are defined using the function prototypes contained in the sqludf.h UDF include file.

```
extern int sqludf_length(
 sqludf_locator* udfloc_p, /* in: User-supplied LOB locator value */
 long* Return_len_p /* out: Return the length of the LOB value */
);
extern int sqludf_substr(
 sqludf_locator* udfloc_p, /* in: User-supplied LOB locator value */
 long start, /* in: Substring start value (starts at 1) */
 long length, /* in: Get this many bytes */
 unsigned char* buffer_p, /* in: Read into this buffer */
 long* Return_len_p /* out: Return the length of the LOB value */
);
extern int sqludf_append(
 sqludf_locator* udfloc_p, /* in: User-supplied LOB locator value */
 unsigned char* buffer_p, /* in: User's data buffer */
 long length, /* in: Length of data to be appended */
 long* Return_len_p /* out: Return the length of the LOB value */
);
extern int sqludf_create_locator(
 int Toc_type, /* in: BLOB, CLOB or DBCLOB? */
 sqludf_locator** Loc_p /* out: Return a ptr to a new locator */
);
```

```

);
extern int sqludf_free_locator(
 sqludf_locator* loc_p /* in: User-supplied LOB locator value */
);

```

The following is a discussion of how these APIs operate. Note that all lengths are in bytes, regardless of the data type, and not in single or double-byte characters.

**Return codes.** Interpret the return code passed back to the UDF by DB2 for each API as follows:

- 0** Success.
- 1** Locator passed to the API was freed by `sqludf_free_locator()` prior to making the call.
- 2** Call was attempted in FENCED mode UDF.
- 3** Bad input value was provided to the API. For examples of bad input values specific to each API, see its description below.
- other** Invalid locator or other error (for example, memory error). The value that is returned for these cases is the SQLCODE corresponding to the error condition. For example, -423 means invalid locator. Please note that before returning to the UDF with one of these "other" codes, DB2 makes a judgement as to the severity of the error. For severe errors, DB2 remembers that the error occurred, and when the UDF returns to DB2, regardless of whether the UDF returns an error SQLSTATE to DB2, DB2 takes action appropriate for the error condition. For non-severe errors, DB2 forgets that the error has occurred, and leaves it up to the UDF to decide whether it can take corrective action, or return an error SQLSTATE to DB2.

- `sqludf_length()`.

Given a LOB locator, it returns the length of the LOB value represented by the locator. The locator in question is generally a locator passed to the UDF by DB2, but could be a locator representing a result value being built (using `sqludf_append()`) by the UDF.

Typically, a UDF uses this API when it wants to find out the length of a LOB value when it receives a locator.

A return code of **3** may indicate:

- `udfloc_p` (address of locator) is zero
- `return_len_p` (address of where to put length) is zero

- `sqludf_substr()`

Given a LOB locator, a beginning position within the LOB, a desired length, and a pointer to a buffer, this API places the bytes into the buffer and returns the number of bytes it was able to move. (Obviously the UDF must provide a buffer large enough for the desired length.) The number of bytes

moved could be shorter than the desired length, for example if you request 50 bytes beginning at position 101 and the LOB value is only 120 bytes long, the API will move only 20 bytes.

Typically, this is the API that a UDF uses when it wants to see the bytes of the LOB value, when it receives a locator.

A return code of **3** may indicate:

- `udfloc_p` (address of locator) is zero
  - `start` is less than 1
  - `length` is negative
  - `buffer_p` (buffer address) is zero
  - `return_len_p` (address of where to put length) is zero
- `sqludf_append()`  
Given a LOB locator, a pointer to a data buffer which has data in it, and a length of data to append, this API appends the data to the end of the LOB value, and returns the length of the bytes appended. (Note that the length appended is always equal to the length given to append. If the entire length cannot be appended, the call to `sqludf_append()` fails with the return code of other.)

Typically, this is the API that a UDF uses when the result is defined with AS LOCATOR, and the UDF is building the result value one append at a time after creating the locator using `sqludf_create_locator()`. After finishing the build process in this case, the UDF moves the locator to where the result argument points.

Note that you can also append to your input locators using this API, which might be useful from the standpoint of maximum flexibility to manipulate your values within the UDF, but this will not have any effect on any LOB values in the SQL statement, or stored in the database.

This API can be used to build very large LOB values in a piecemeal manner. In cases where a large number of appends is used to build a result, the performance of this task can be improved by:

- allocating a large application control heap (`APP_CTL_HEAP_SZ` is the database manager configuration parameter)
- doing fewer appends of larger buffers; for example, instead of doing 20 appends of 50 bytes each, doing a single 1000 byte append

SQL applications which build many large LOB values via the `sqludf_append()` API may encounter errors caused by limitations on the amount of disk space available. The chance of these errors happening can be reduced by:

- using larger buffers for the individual appends
- doing frequent COMMITs between statements

- in cases where each row of a SELECT statement is building a LOB value via this API, using a CURSOR WITH HOLD and doing COMMITs between rows

A return code of 3 may indicate:

- `udfloc_p` (address of locator) is zero
- length is negative
- `buffer_p` (buffer address) is zero

- `sqludf_create_locator()`

Given a data type, for example `SQL_TYP_CLOB`, it creates a locator. (The data type values are defined in the external application header file `sql.h`.)

Typically, a UDF uses this API when the UDF result is defined with `AS LOCATOR`, and the UDF wants to build the result value using `sqludf_append()`. Another use is to internally manipulate LOB values.

A return code of 3 may indicate:

- `udfloc_p` (address of locator) is zero
- `loc_type` is not one of the three valid values
- `loc_p` (address of where to put locator) is zero

- `sqludf_free_locator()`

Frees the passed locator.

Use this API to free any locators that were created with the `sqludf_create_locator()` API, and which were used only for internal manipulation. It is NOT NECESSARY to free locators passed into the UDF. It is NOT NECESSARY to free any locator created by the UDF via `sqludf_create_locator()` if that locator is passed out of the UDF as an output.

A return code of 3 may indicate:

- `udfloc_p` (address of locator) is zero

The following notes apply to the use of these APIs:

**Notes:**

1. A UDF which is defined to return a LOB locator has several possibilities available to it. It can return:
  - an input locator passed to it
  - an input locator passed to it which has been appended to via `sqludf_append()`
  - a locator created to via `sqludf_create_locator()`, and appended to via `sqludf_append()`
2. A table function can be defined as returning one or more LOB locators. Each of them can be any of the possibilities discussed in the preceding

item. It is also valid for such a table function to return the same locator as an output for several table function columns.

3. A LOB locator passed to a table function as an input argument remains alive for the entire duration of the row generation process. In fact, the table function can append to a LOB using such a LOB locator while generating one row, and see the appended bytes on a subsequent row.
4. The internal control mechanisms used to represent a LOB which originated in DB2 as a LOB locator output from a UDF (table or scalar function), take 1950 bytes. For this reason, and because there are limitations on the size of a row which is input to a sort, a query which attempts to sort multiple such LOBs which originated as UDF LOB locators will be limited to (at most) two such values per row, depending on the sizes of the other columns involved. The same limitation applies to rows being inserted into a table.

## Scenarios for Using LOB Locators

This is a brief summary of possible scenarios that show the usefulness of LOB locators. These four scenarios outline the use of locators, and show how you can reduce space requirements and increase efficiency.

- Varying access to parts of an input LOB.

A UDF looks at the first part of a LOB value using `sqludf_substr()`, and based on a size variable it finds there, it may want to read just a few bytes from anywhere in the 100 million byte LOB value, again using `sqludf_substr()`.

- Process most of an input LOB one part at a time.

This UDF is looking for something in the LOB value. Most often it will find it near the front, but sometimes it may have to scan the entire 100 million byte value. The UDF uses `sqludf_length()` to find the size of this particular value, and steps through the value 1 000 bytes at a time by placing a call to `sqludf_substr()` in a loop. It uses a variable as the starting position, increasing the variable by 1 000 each time through the loop. It proceeds in this manner until it finds what it is looking for.

- Return one of the two input LOBs

This UDF has two LOB locators as inputs, and returns a LOB locator as an output. It examines and compares the two inputs, reading the bytes received using `sqludf_substr()` and then determines which of the two to select based on some algorithm. When it determines this, it copies the locator of the selected input to the buffer indicated by the UDF result argument, and exits.

- Cut and paste an input LOB, and return the result.

The UDF is passed a LOB value and maybe some other arguments which presumably tell it how to proceed. It creates a locator for its output, and proceeds to build the output value sequentially, taking most of the result

value from different parts of the input LOB which it reads using `sqludf_substr()`, based on the instructions contained in the other input arguments. Finally when it is done it copies the result locator to the buffer to which the UDF result argument points, and then exits.

---

## Other Coding Considerations

This section documents additional considerations for implementing a UDF, items to keep in mind, and items to avoid.

### Hints and Tips

The following are recommendations to consider to successfully implement your UDF:

- **UDF bodies need to be protected.** The executable function bodies are not captured or protected in any way by DB2. The `CREATE FUNCTION` statement merely points to the body. To preserve the integrity of the function and the database applications which depend on the function, you must, by managing access to the directory containing the function and by protecting the body itself, prevent the function body from being inadvertently or intentionally deleted or replaced.
- DB2 passes pointers to all of the buffers in the interface between DB2 and SQL (that is, all the SQL arguments and the function return value). Be sure you define your UDF arguments as pointers.
- All SQL argument values are buffered. This means that a copy of the value is made and presented to the UDF. If a UDF changes its input parameters, the changes have no effect on SQL values or processing, but may cause DB2 to malfunction.
- For OLE automation, do not change the input parameters, otherwise memory resources may not be freed and you may encounter a memory leak.

In case of a major OLE library version mismatch or a failure in initializing the OLE library, the database manager returns SQLCODE -465 (SQLSTATE 58032) with reason code 34, (Failure to initialize OLE library).

- Re-entrancy is **strongly recommended** for UDFs on all operating platforms, so that one copy of it can be used for multiple concurrent statements and applications.

Note that the SCRATCHPAD facility can be used to circumvent many of the limitations imposed by re-entrancy.

- If the body of a function currently being used is modified (for example, recompiled and relinked), DB2 will not *change functions* in mid-transaction. However, the copy used in a subsequent transaction may be different if this



kind of dynamic modification is taking place. Your operating system may also prevent you from changing a UDF body that is in use. This practice is not recommended.

- If you allocate dynamic memory in the UDF, it should be freed before returning to DB2. This is especially important for the NOT FENCED case. The SCRATCHPAD facility can be used, however, to anchor dynamic memory needed by the UDF across invocations. If you use the scratchpad in this manner, specify the FINAL CALL attribute on the CREATE FUNCTION for the UDF so that it can free the allocated memory at end-of-statement processing. The reason for this is that the system could run out of memory over time, with repeated use of the UDF.  
This reasoning holds as well for other system resources used by the UDF.
- Use the NOT NULL CALL option if it makes sense to do so. With this CREATE FUNCTION option, you do not have to check whether each SQL argument is null, and it performs better when you do have NULL values.
- Use the NOT DETERMINISTIC option if the result from your UDF depends on anything other than the input SQL arguments. This option prevents the SQL compiler from performing certain optimizations which can cause inconsistent results.
- Use the EXTERNAL ACTION option if your UDF has any side effects which need to be reliably performed. EXTERNAL ACTION prevents the SQL compiler from performing certain optimizations which can prevent invocation of your UDF in certain circumstances.
- Use the FENCED option, unless you are either:
  - Working on a test database under circumstances where the integrity of the database does not matter
  - Absolutely certain that the UDF can perform no potentially damaging actions resulting in an erroneous modification of storage. *DB2 provides some protection against these kinds of actions, but does not guarantee the integrity of the database if you run NOT FENCED.* See “Debugging your UDF” on page 371 for more information on running FENCED.
- For considerations on using UDFs with EUC code sets, see “Considerations for UDFs” on page 423.
- For an application running NOT FENCED UDFs, the first time such a UDF is invoked, a block of memory of the size indicated by the UDF\_MEM\_SZ configuration parameter is created. Thereafter, on a statement by statement basis, memory for interfacing between DB2 and NOT FENCED UDFs is allocated and deallocated from this block of memory as needed.

For FENCED UDFs, a different block of memory is used in the same way. It is different because the memory is shared between processes. In fact, if an application uses both NOT FENCED and FENCED UDFs, two separate

blocks of memory, each of the size indicated by the UDF\_MEM\_SZ parameter are used. Refer to the *Administration Guide* for more information about this configuration parameter.

- Use the DISALLOW PARALLELISM option in the following situations:
  - On scalar UDFs, if your UDF absolutely depends on running the same copy. Generally, this will be the case for NOT DETERMINISTIC SCRATCHPAD UDFs. (For an example, see the counter UDF specified in “Scratchpad Considerations” on page 331.)
  - If you do not want the UDF to run on multiple partitions at once for a single reference.
  - If you are specifying a table function.

Otherwise, ALLOW PARALLELISM (the default) should be specified.

## UDF Restrictions and Caveats

This section discusses items to be avoided in your UDF:

1. In general DB2 does not restrict the use of operating system functions. A few exceptions are:
  - a. Registering of signal or exception handlers may interfere with DB2's use of these same handlers and may result in unexpected failure.
  - b. System calls that terminate a process may abnormally terminate one of DB2's processes and result in system or application failure.

Other system calls may also cause problems if they interfere with the normal operation of DB2; for example, a UDF that attempts to unload a library containing a UDF from memory could cause severe problems. Be careful in coding and testing any UDFs containing system calls.
2. The values of all environment variables beginning with 'DB2' are captured at the time the database manager is started with db2start, and are available in all UDFs whether or not they are FENCED. The only exception is the DB2CKPTR environment variable. Note that the environment variables are *captured*; any changes to the environment variables after db2start is issued are not available to the UDFs.
3. With respect to LOBs passed to an external UDF, you are limited to the maximum size specified by the *UDF Shared Memory Size* DB2 system configuration parameter. The maximum that you can specify for this parameter is 256M. The default setting on DB2 is 1M. For more information on this parameter, refer to the *Administration Guide*.
4. Input to, and output from, the screen and keyboard is not recommended. In the process model of DB2, UDFs run in the background, so you cannot write to the screen. However, you can write to a file.

**Note:** DB2 does not attempt to synchronize any external input/output performed by a UDF with DB2's own transactions. So for example,

if a UDF writes to a file during a transaction, and that transaction is later backed out for some reason, no attempt is made to discover or undo the writes to the file.

5. On UNIX-based systems, your UDF runs under the UID of the DB2 Agent Process (NOT FENCED), or the UID which owns the db2udf executable (FENCED). This UID controls the system resources available to the UDF. For information on the db2udf executable, refer to the *Quick Beginnings* for your platform.
6. When using protected resources, (that is, resources that only allow one process access at a time) inside UDFs, you should try to avoid deadlocks between UDFs. If two or more UDFs deadlock, DB2 will not be able to detect the condition.
7. Character data is passed to external functions in the code page of the database. Likewise, a character string that is output from the function is assumed by the database to use the database's code page. In the case where the application code page differs from the database code page, the code page conversions occur as they would for other values in the SQL statement. You can prevent this conversion, by coding FOR BIT DATA as an attribute of the character parameter or result in your CREATE FUNCTION statement. If the character parameter is not defined with the FOR BIT DATA attribute, your UDF code will receive arguments in the database code page.

Note that, using the DBINFO option on CREATE FUNCTION, the database code page is passed to the UDF. Using this information, a UDF which is sensitive to the code page can be written to operate in many different code pages.

8. When writing a UDF using C++, you may want to consider declaring the function name as:

```
extern "C" void SQL_API_FN udf(...arguments...)
```

The extern "C" prevents type decoration (or 'mangling') of the function name by the C++ compiler. Without this declaration, you have to include all the type decoration for the function name when you issue the CREATE FUNCTION statement.

---

# Examples of UDF Code

The following UDF code examples are supplied with DB2.

- Example: Integer Divide Operator
- Example: Fold the CLOB, Find the Vowel
- Example: Counter

For information on where to find all the examples supplied, and how to invoke them, see “Appendix B. Sample Programs” on page 633.

For information on compiling and linking UDFs, refer to the *Application Building Guide*.

Each of the example UDFs is accompanied by the corresponding CREATE FUNCTION statement, and a small scenario showing its use. These scenarios all use the following table TEST, which has been carefully crafted to illustrate certain points being made in the scenarios. Here is the table definition:

```
CREATE TABLE TEST (INT1 INTEGER,
 INT2 INTEGER,
 PART CHAR(5),
 DESCR CLOB(33K))
```

After populating the table, issue the following statement using CLP to display its contents:

```
SELECT INT1, INT2, PART, SUBSTR(DESCR,1,50) FROM TEST
```

Note the use of the SUBSTR function on the CLOB column to make the output more readable. You receive the following CLP output:

| INT1                  | INT2 | PART     | 4                                                |
|-----------------------|------|----------|--------------------------------------------------|
| -----                 |      |          |                                                  |
| 16                    |      | 1 brain  | The only part of the body capable of forgetting. |
| 8                     |      | 2 heart  | The seat of the emotions?                        |
| 4                     |      | 4 elbow  | That bendy place in mid-arm.                     |
| 2                     |      | 0 -      | -                                                |
| 97                    |      | 16 xxxxx | Unknown.                                         |
| 5 record(s) selected. |      |          |                                                  |

Refer to the previous information on table TEST as you read the examples and scenarios which follow.

## Example: Integer Divide Operator

Suppose you are unhappy with the way integer divide works in DB2 because it returns an error, SQLCODE -802 (SQLSTATE 22003), and terminates the statement when the divisor is zero. (Note that if you enable *friendly arithmetic* with the DFT\_SQLMATHWARN configuration parameter, DB2 returns a

NULL instead of an error in this situation.) Instead, you want the integer divide to return a NULL, so you code this UDF:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqllda.h>
#include "util.h"

/*****
 * function divid: performs integer divid, but unlike the / operator
 * shipped with the product, gives NULL when the
 * denominator is zero.
 *
 * This function does not use the constructs defined in the
 * "sqludf.h" header file.
 *
 * inputs: INTEGER num numerator
 * INTEGER denom denominator
 * output: INTEGER out answer
 *****/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN divid (
 long *num, /* numerator */
 long *denom, /* denominator */
 long *out, /* output result */
 short *in1null, /* input 1 NULL indicator */
 short *in2null, /* input 2 NULL indicator */
 short *outnull, /* output NULL indicator */
 char *sqlstate, /* SQL STATE */
 char *funcname, /* function name */
 char *specname, /* specific function name */
 char *mesgtext) { /* message text insert */

 if (*denom == 0) { /* if denominator is zero, return null result */
 *outnull = -1;
 } else { /* else, compute the answer */
 *out = *num / *denom;
 *outnull = 0;
 } /* endif */
}
/* end of UDF : divid */
```

For this UDF, notice that:

- It does not include `sqludf.h`.
- It has two input arguments defined, and one output argument.
- It is defined to return `void`. Remember that the normal UDF outputs will be returned using the input arguments.

- The inclusion of SQL\_API\_FN in the function definition is designed to assure portability of function source across platforms. It requires the inclusion of the following statement in your UDF source files.  

```
#include <sqlsystem.h>
```
- It does not check for null input arguments, because the NOT NULL CALL parameter is specified by default in the CREATE FUNCTION statement shown below.

Here is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION MATH."/"(INT,INT)
 RETURNS INT
 NOT FENCED
 DETERMINISTIC
 NO SQL
 NO EXTERNAL ACTION
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 EXTERNAL NAME '/u/slick/udfx/div' ;
```

(This statement is for an AIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.)

For this statement, observe that:

- It is defined to be in the MATH schema. In order to define a UDF in a schema that is not equal to your user-ID, you need DBADM authority on the database.
- The function name is defined to be "/", the same name as the SQL divide operator. In fact, this UDF can be invoked the same as the built-in / operator, using either *infix notation*, for example, A / B, or *functional notation*, for example, "/"(A,B). See below.
- You have chosen to define it as NOT FENCED because you are absolutely sure that the program has no errors.
- You have used the default NOT NULL CALL, by which DB2 provides a NULL result if either argument is NULL, without invoking the body of the function.

Now if you run the following pair of statements (CLP input is shown):

```
SET CURRENT FUNCTION PATH = SYSIBM, SYSFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

You get this output from CLP (if you do not enable *friendly arithmetic* with the database configuration parameter DFT\_SQLMATHWARN):

| INT1 | INT2 | 3  | 4  |
|------|------|----|----|
| 16   | 1    | 16 | 16 |
| 8    | 2    | 4  | 4  |
| 4    | 4    | 1  | 1  |

SQL0802N Arithmetic overflow or other arithmetic exception occurred.  
SQLSTATE=22003

The SQL0802N error message occurs because you have set your CURRENT FUNCTION PATH special register to a concatenation of schemas which does not include MATH, the schema in which the "/" UDF is defined. And therefore you are executing DB2's built-in divide operator, whose defined behavior is to give the error when a "divide by zero" condition occurs. The fourth row in the TEST table provides this condition.

However, if you change the function path, putting MATH in front of SYSIBM in the path, and rerun the SELECT statement:

```
SET CURRENT FUNCTION PATH = MATH, SYSIBM, SYSPFUN, SLICK
SELECT INT1, INT2, INT1/INT2, "/"(INT1,INT2) FROM TEST
```

You then get the desired behavior, as shown by the following CLP output:

| INT1 | INT2 | 3  | 4  |
|------|------|----|----|
| 16   | 1    | 16 | 16 |
| 8    | 2    | 4  | 4  |
| 4    | 4    | 1  | 1  |
| 2    | 0    | -  | -  |
| 97   | 16   | 6  | 6  |

5 record(s) selected.

For the above example, observe that:

- The SET CURRENT FUNCTION PATH statement changes the current function path used in the following statement as this is dynamic SQL, placing the MATH schema ahead of SYSIBM.
- The fourth row produces a NULL result from the divides, and the statement continues.
- Both syntaxes (*infix syntax*, and *prefix syntax*) can be used to invoke this particular UDF, because its name is the same as a built-in operator, and both are used in the above example, with identical results.
- As a practical note, because of the way the built-in functions and operators are defined to DB2, this "/" is not used for operations on SMALLINTs. The DB2 function selection algorithm chooses the exact match built-in "/" operator in preference to this user-defined "/", which is a match but not an exact match. There are different ways around this seeming inconsistency. You can explicitly cast SMALLINT arguments to INTEGER before invoking

`"/`, for example, `INT1 / INTEGER(SMINT1)` (where the column `SMINT1` is assumed to be `SMALLINT`). Or, better than that, you could register additional UDFs, further overloading the `"/` operator, which define first and second parameters that are `SMALLINT`. These additional UDFs can be sourced on `MATH. "/"`.

In this case, for a fully general set of functions you have to `CREATE` the following three additional functions to completely handle integer divide:

```
CREATE FUNCTION MATH. "/" (SMALLINT,SMALLINT)
 RETURNS INT
 SOURCE MATH. "/" (INT,INT)

CREATE FUNCTION MATH. "/" (SMALLINT,INT)
 RETURNS INT
 SOURCE MATH. "/" (INT,INT)

CREATE FUNCTION MATH. "/" (INT,SMALLINT)
 RETURNS INT
 SOURCE MATH. "/" (INT,INT)
```

Even though three UDFs are added, additional code does not have to be written as they are sourced on `MATH. "/"`.

And now, with the definition of these four `"/` functions, any users who want to take advantage of the new behavior on integer divide need only place `MATH` ahead of `SYSIBM` in their function path, and can write their SQL as usual.

While the preceding example does not consider the `BIGINT` data type, you can easily extend the example to include `BIGINT`.

### Example: Fold the CLOB, Find the Vowel

Suppose you have coded up two UDFs to help you with your text handling application. The first UDF *folds* your text string after the *n*th byte. In this example, fold means to put the part that was originally after the *n* byte before the part that was originally in front of the *n+1* byte. In other words, the UDF moves the first *n* bytes from the beginning of the string to the end of the string. The second function returns the position of the first vowel in the text string. Both of these functions are coded in the `udf.c` example file:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqllda.h>
#include "util.h"
```



```

/*****
* function fold: input string is folded at the point indicated by the
* second argument.
*
* input: CLOB in1 input string
* INTEGER in2 position to fold on
* CLOB out folded string
*****/
#ifdef _cplusplus
extern "C"
#endif
void SQL_API_FN fold (
 SQLUDF_CLOB *in1, /* input CLOB to fold */
 SQLUDF_INTEGER *in2, /* position to fold on */
 SQLUDF_CLOB *out, /* output CLOB, folded */
 SQLUDF_NULLIND *in1null, /* input 1 NULL indicator */
 SQLUDF_NULLIND *in2null, /* input 2 NULL indicator */
 SQLUDF_NULLIND *outnull, /* output NULL indicator */
 SQLUDF_TRAIL_ARGS) { /* trailing arguments */

 SQLUDF_INTEGER len1;

 if (SQLUDF_NULL(in1null) || SQLUDF_NULL(in2null)) {
 /* one of the arguments is NULL. The result is then "INVALID INPUT" */
 strcpy((char *) out->data, "INVALID INPUT");
 out->length = strlen("INVALID INPUT");
 } else {
 len1 = in1->length; /* length of the CLOB */

 /* build the output by folding at position "in2" */
 strncpy((char *) out->data, &in1->data[*in2], len1 - *in2);
 strncpy((char *) &out->data[len1 - *in2], in1->data, *in2);
 out->length = in1->length;
 } /* endif */
 outnull = 0; / result is always non-NULL */
}
/* end of UDF : fold */

/*****
* function findvwl: returns the position of the first vowel.
*
* returns an error if no vowel is found
* when the function is created, must be defined as
* NOT NULL CALL.
* inputs: VARCHAR(500) in
* output: INTEGER out
*****/
#ifdef _cplusplus
extern "C"
#endif
void SQL_API_FN findvwl (
 SQLUDF_VARCHAR *in, /* input character string */
 SQLUDF_SMALLINT *out, /* output location of vowel */
 SQLUDF_NULLIND *innull, /* input NULL indicator */
 SQLUDF_NULLIND *outnull, /* output NULL indicator */
 SQLUDF_TRAIL_ARGS) { /* trailing arguments */

```

```

short i; /* local indexing variable */

for (i=0; (i < (short)strlen(in) && /* find the first vowel */
 in[i] != 'a' && in[i] != 'e' && in[i] != 'i' &&
 in[i] != 'o' && in[i] != 'u' && in[i] != 'y' &&
 in[i] != 'A' && in[i] != 'E' && in[i] != 'I' &&
 in[i] != 'O' && in[i] != 'U' && in[i] != 'Y'); i++);
if (i == strlen((char *) in)) { /* no vowels found */
 strcpy((char *) sqludf_sqlstate, "38999"); /* error state */
 strcpy((char *) sqludf_msgtext, "findvwl: No Vowel");
 /* message insert */
} else { /* a vowel was found at "i" */
 *out = i + 1;
 *outnull = 0;
} /* endif */
}
/* end of UDF : findvwl */

```

For the above UDFs, notice:

- They include `sqludf.h`, and use the argument definitions and macros contained in that file.
- The `fold()` function is invoked even with NULL arguments, and returns the string `INVALID INPUT` in this case. The `findvwl()` function on the other hand, is not invoked with null arguments. The use of the `SQLUDF_NULL()` macro, defined in `sqludf.h` checks for null arguments in `fold()`.
- The `findvwl()` function sets the error `SQLSTATE` and the message token.
- The `fold()` function returns a CLOB value in addition to having the CLOB data type as its text input argument. The `findvwl()` has a `VARCHAR` input argument.

Here are the `CREATE FUNCTION` statements for these UDFs:

```

CREATE FUNCTION FOLD(CLOB(100K),INT)
 RETURNS CLOB(100K)
 FENCED
 DETERMINISTIC
 NO SQL
 NO EXTERNAL ACTION
 LANGUAGE C
 NULL CALL
 PARAMETER STYLE DB2SQL
 EXTERNAL NAME 'udf!fold' ;

CREATE FUNCTION FINDV(VARCHAR(500))
 RETURNS INTEGER
 NOT FENCED
 DETERMINISTIC
 NO SQL
 NO EXTERNAL ACTION

```

```

LANGUAGE C
NOT NULL CALL
PARAMETER STYLE DB2SQL
EXTERNAL NAME 'udf!findvwl' ;

```

The above CREATE FUNCTION statements are for UNIX-based platforms. On other platforms, you may need to modify the value specified in the EXTERNAL NAME clause in the above statements. You can find the above CREATE FUNCTION statements in the calludf.sqc example program shipped with DB2.

Referring to these CREATE statements, observe that:

- The schema names of the functions will default to the statement authorization-ID.
- You have chosen to define FOLD as FENCED because you are not absolutely sure that it is error-free, but FINDV is NOT FENCED.
- You have coded NULL CALL for FOLD, which means that fold() will be called even if either input argument is null, which agrees with the way the function is coded. FINDV is coded to be NOT NULL CALL, which also agrees with the code.
- Both will default to ALLOW PARALLELISM.

Now you can successfully run the following statement:

```
SELECT SUBSTR(DESCR,1,30), SUBSTR(FOLD(DESCR,6),1,30) FROM TEST
```

The output from the CLP for this statement is:

| 1                              | 2                              |
|--------------------------------|--------------------------------|
| -----                          | -----                          |
| The only part of the body capa | ly part of the body capable of |
| The seat of the emotions?      | at of the emotions?The se      |
| That bendy place in mid-arm.   | endy place in mid-arm.That b   |
| -                              | INVALID INPUT                  |
| Unknown.                       | n.Unknow                       |
| 5 record(s) selected.          |                                |

Note the use of the SUBSTR built-in function to make the selected CLOB values display more nicely. It shows how the output is folded (best seen in the second, third and fifth rows, which have a shorter CLOB value than the first row, and thus the folding is more evident even with the use of SUBSTR). And it shows (fourth row) how the INVALID INPUT string is returned by the FOLD UDF when its input text string (column DESCR) is null. This SELECT also shows simple nesting of function references; the reference to FOLD is within an argument of the SUBSTR function reference.

Then if you run the following statement:

```
SELECT PART, FINDV(PART) FROM TEST
```

The CLP output is as follows:

```
PART 2

brain 3
heart 2
elbow 1
- -
SQL0443N User defined function "SLICK.FINDV" (specific name
"SQL950424135144750") has returned an error SQLSTATE with diagnostic
text "findvwl: No Vowel". SQLSTATE=38999
```

This example shows how the 38999 SQLSTATE value and error message token returned by `findvwl()` are handled: message SQL0443N returns this information to the user. The PART column in the fifth row contains no vowel, and this is the condition which triggers the error in the UDF.

Observe the argument promotion in this example. The PART column is CHAR(5), and is promoted to VARCHAR to be passed to FINDV.

And finally note how DB2 has generated a null output from FINDV for the fourth row, as a result of the NOT NULL CALL specification in the CREATE statement for FINDV.

The following statement:

```
SELECT SUBSTR(DESCR,1,25), FINDV(CAST (DESCR AS VARCHAR(60)))
FROM TEST
```

Produces this output when executed in the CLP:

```
1 2

The only part of the body 3
The seat of the emotions? 3
That bendy place in mid-a 3
- -
Unknown. 1
5 record(s) selected.
```

This SELECT statement shows FINDV working on a VARCHAR input argument. Observe how we cast column DESCR to VARCHAR to make this happen. Without the cast we would not be able to use FINDV on a CLOB argument, because CLOB is not promotable to VARCHAR. Again, the built-in SUBSTR function is used to make the DESCR column value display better.

And here again note that the fourth row produces a null result from FINDV because of the NOT NULL CALL.

## Example: Counter

Suppose you want to simply number the rows in your `SELECT` statement. So you write a UDF which increments and returns a counter. This UDF uses a scratchpad:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h>
#include <sqlca.h>
#include <sqlda.h>
#include "util.h"

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
 long len;
 long countr;
 char not_used[96];
} ;

/*****
 * function ctr: increments and reports the value from the scratchpad.
 *
 * This function does not use the constructs defined in the
 * "sqludf.h" header file.
 *
 * input: NONE
 * output: INTEGER out the value from the scratchpad
 *****/
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN ctr (
 long *out, /* output answer (counter) */
 short *outnull, /* output NULL indicator */
 char *sqlstate, /* SQL STATE */
 char *funcname, /* function name */
 char *specname, /* specific function name */
 char *mesgtext, /* message text insert */
 struct scr *scratchptr) { /* scratch pad */

 out = ++scratchptr->countr; / increment counter & copy out */
 *outnull = 0;
}
/* end of UDF : ctr */
```

For this UDF, observe that:

- It does not include `sqludf.h`. But it does include `sqlsystem.h` for the definition of `SQL_API_FN`.
- It has no input SQL arguments defined, but returns a value.
- It appends the scratchpad input argument after the four standard trailing arguments, namely *SQL-state*, *function-name*, *specific-name*, and *message-text*.

- It includes a structure definition to map the scratchpad which is passed.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION COUNTER()
 RETURNS INT
 SCRATCHPAD
 NOT FENCED
 NOT DETERMINISTIC
 NO SQL
 NO EXTERNAL ACTION
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 EXTERNAL NAME 'udf!ctr'
 DISALLOW PARALLELISM;
```

(This statement is for an AIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.)

Referring to this statement, observe that:

- No input parameters are defined. This agrees with the code.
- SCRATCHPAD is coded, causing DB2 to allocate, properly initialize and pass the scratchpad argument.
- You have chosen to define it as NOT FENCED because you are absolutely sure that it is error free.
- You have specified it to be NOT DETERMINISTIC, because it depends on more than the SQL input arguments, (none in this case).
- You have correctly specified DISALLOW PARALLELISM, because correct functioning of the UDF depends on a single scratchpad.

Now you can successfully run the following statement:

```
SELECT INT1, COUNTER(), INT1/COUNTER() FROM TEST
```

When run through the CLP, it produces this output:

| INT1 | 2 | 3  |
|------|---|----|
| 16   | 1 | 16 |
| 8    | 2 | 4  |
| 4    | 3 | 1  |
| 2    | 4 | 0  |
| 97   | 5 | 19 |

5 record(s) selected.

Observe that the second column shows the straight COUNTER() output. The third column shows that the two separate references to COUNTER() in the

SELECT statement each get their own scratchpad; had they not each gotten their own, the output in the second column would have been 1 3 5 7 9, instead of the nice orderly 1 2 3 4 5.

### Example: Weather Table Function

The following is an example table function, `tfweather_u`, (supplied by DB2 in the programming example `tblsrv.c`), that returns weather information for various cities in the United States. The weather data for these cities is included in the example program, but could be read in from an external file, as indicated in the comments contained in the example program. The data includes the name of a city followed by its weather information. This pattern is repeated for the other cities. Note that there is a client application (`tblcli.sqc`) supplied with DB2 that calls this table function and prints out the weather data retrieved using the `tfweather_u` table function.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqludf.h> /* for use in compiling User Defined Function */

#define SQL_NOTNULL 0 /* Nulls Allowed - Value is not Null */
#define SQL_ISNULL -1 /* Nulls Allowed - Value is Null */

/* Short and long city name structure */
typedef struct {
 char * city_short ;
 char * city_long ;
} city_area ;

/* Scratchpad data */
/* Preserve information from one function call to the next call */
typedef struct {
 /* FILE * file_ptr; if you use weather data text file */
 int file_pos ; /* if you use a weather data buffer */
} scratch_area ;

/* Field descriptor structure */
typedef struct {
 char fld_field[31] ; /* Field data */
 int fld_ind ; /* Field null indicator data */
 int fld_type ; /* Field type */
 int fld_length ; /* Field length in the weather data */
 int fld_offset ; /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
 { "alb", "Albany, NY" },
 { "atl", "Atlanta, GA" },
 .
 .
 .
}
```

```

 { "wbc", "Washington DC, DC" },
 /* You may want to add more cities here */

 /* Do not forget a null termination */
 { (char *) 0, (char *) 0 }
} ;

/* Field descriptor data */
fld_desc fields[] = {
 { "", SQL_ISNULL, SQL_TYP_VARCHAR, 30, 0 }, /* city */
 { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 2 }, /* temp_in_f */
 { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 7 }, /* humidity */
 { "", SQL_ISNULL, SQL_TYP_VARCHAR, 5, 13 }, /* wind */
 { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 19 }, /* wind_velocity */
 { "", SQL_ISNULL, SQL_TYP_FLOAT, 5, 24 }, /* barometer */
 { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast */
 /* You may want to add more fields here */

 /* Do not forget a null termination */
 { (char) 0, 0, 0, 0, 0 }
} ;

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement. Note that you */
/* have to specify the full path name for this file. */
char * weather_data[] = {
 "alb.forecast",
 " 34 28% wnw 3 30.53 clear",
 "atl.forecast",
 " 46 89% east 11 30.03 fog",
 .
 .
 .
 "wbc.forecast",
 " 38 96% ene 16 30.31 light rain",
 /* You may want to add more weather data here */

 /* Do not forget a null termination */
 (char *) 0
} ;

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Find a full city name using a short name */
int get_name(char * short_name, char * long_name) {

 int name_pos = 0 ;

 while (cities[name_pos].city_short != (char *) 0) {
 if (strcmp(short_name, cities[name_pos].city_short) == 0) {
 strcpy(long_name, cities[name_pos].city_long) ;

```



```

 /* A full city name found */
 return(0) ;
 }
 name_pos++ ;
}
/* Could not find such city in the city data */
strcpy(long_name, "Unknown City") ;
return(-1) ;
}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Clean all field data and field null indicator data */
int clean_fields(int field_pos) {

 while (fields[field_pos].fld_length != 0) {
 memset(fields[field_pos].fld_field, '\0', 31) ;
 fields[field_pos].fld_ind = SQL_ISNULL ;
 field_pos++ ;
 }
 return(0) ;
}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value(char * value, int field_pos) {

 fld_desc * field ;
 char field_buf[31] ;
 double * double_ptr ;
 int * int_ptr, buf_pos ;

 while (fields[field_pos].fld_length != 0) {
 field = &fields[field_pos] ;
 memset(field_buf, '\0', 31) ;
 memcpy(field_buf,
 (value + field->fld_offset),
 field->fld_length) ;
 buf_pos = field->fld_length ;
 while ((buf_pos > 0) &&
 (field_buf[buf_pos] == ' '))
 field_buf[buf_pos--] = '\0' ;
 buf_pos = 0 ;
 while ((buf_pos < field->fld_length) &&
 (field_buf[buf_pos] == ' '))
 buf_pos++ ;
 if (strlen((char *) (field_buf + buf_pos)) > 0 ||

```

```

 strcmp((char *) (field_buf + buf_pos), "n/a") != 0) {
 field->fld_ind = SQL_NOTNULL;

 /* Text to SQL type conversion */
 switch(field->fld_type) {
 case SQL_TYP_VARCHAR:
 strcpy(field->fld_field,
 (char *) (field_buf + buf_pos));
 break;
 case SQL_TYP_INTEGER:
 int_ptr = (int *) field->fld_field;
 *int_ptr = atoi((char *) (field_buf + buf_pos));
 break;
 case SQL_TYP_FLOAT:
 double_ptr = (double *) field->fld_field;
 *double_ptr = atof((char *) (field_buf + buf_pos));
 break;
 /* You may want to add more text to SQL type conversion here */
 }

 }
 field_pos++;
}
return(0);
}

#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN weather(/* Return row fields */
 SQLUDF_VARCHAR * city,
 SQLUDF_INTEGER * temp_in_f,
 SQLUDF_INTEGER * humidity,
 SQLUDF_VARCHAR * wind,
 SQLUDF_INTEGER * wind_velocity,
 SQLUDF_DOUBLE * barometer,
 SQLUDF_VARCHAR * forecast,
 /* You may want to add more fields here */

 /* Return row field null indicators */
 SQLUDF_NULLIND * city_ind,
 SQLUDF_NULLIND * temp_in_f_ind,
 SQLUDF_NULLIND * humidity_ind,
 SQLUDF_NULLIND * wind_ind,
 SQLUDF_NULLIND * wind_velocity_ind,
 SQLUDF_NULLIND * barometer_ind,
 SQLUDF_NULLIND * forecast_ind,
 /* You may want to add more field indicators here */

 /* UDF always-present (trailing) input arguments */
 SQLUDF_TRAIL_ARGS_ALL
) {

 scratch_area * save_area ;

```

```

char line_buf[81] ;
int line_buf_pos ;

/* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
/* Preserve information from one function call to the next call */
save_area = (scratch_area *) (SQLUDF_SCRAT->data) ;

/* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
switch(SQLUDF_CALLT) {

 /* First call UDF: Open table and fetch first row */
 case SQL_TF_OPEN:
 /* If you use a weather data text file specify full path */
 /* save_area->file_ptr = fopen("/sqllib/samples/c/tblsrv.dat",
 "r"); */

 save_area->file_ptr = 0 ;
 break ;

 /* Normal call UDF: Fetch next row */
 case SQL_TF_FETCH:
 /* If you use a weather data text file */
 /* memset(line_buf, '\0', 81); */
 /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
 if (weather_data[save_area->file_ptr] == (char *) 0) {

 /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
 strcpy(SQLUDF_STATE, "02000") ;

 break ;
 }
 memset(line_buf, '\0', 81) ;
 strcpy(line_buf, weather_data[save_area->file_ptr]) ;
 line_buf[3] = '\0' ;

 /* Clean all field data and field null indicator data */
 clean_fields(0) ;

 /* Fills city field null indicator data */
 fields[0].fld_ind = SQL_NOTNULL ;

 /* Find a full city name using a short name */
 /* Fills city field data */
 if (get_name(line_buf, fields[0].fld_field) == 0) {
 save_area->file_ptr++ ;
 /* If you use a weather data text file */
 /* memset(line_buf, '\0', 81); */
 /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
 if (weather_data[save_area->file_ptr] == (char *) 0) {
 /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
 strcpy(SQLUDF_STATE, "02000") ;
 break ;
 }
 memset(line_buf, '\0', 81) ;
 strcpy(line_buf, weather_data[save_area->file_ptr]) ;
 line_buf_pos = strlen(line_buf) ;
 }
 }
}

```

```

while (line_buf_pos > 0) {
 if (line_buf[line_buf_pos] >= ' ')
 line_buf_pos = 0 ;
 else {
 line_buf[line_buf_pos] = '\0' ;
 line_buf_pos-- ;
 }
}

/* Fills field data and field null indicator data ... */
/* ... for selected city from text weather data */
get_value(line_buf, 1) ; /* Skips city field */

/* Builds return row fields */
strcpy(city, fields[0].fld_field) ;
memcpy((void *) temp_in_f,
 fields[1].fld_field,
 sizeof(SQLUDF_INTEGER)) ;
memcpy((void *) humidity,
 fields[2].fld_field,
 sizeof(SQLUDF_INTEGER)) ;
strcpy(wind, fields[3].fld_field) ;
memcpy((void *) wind_velocity,
 fields[4].fld_field,
 sizeof(SQLUDF_INTEGER)) ;
memcpy((void *) barometer,
 fields[5].fld_field,
 sizeof(SQLUDF_DOUBLE)) ;
strcpy(forecast, fields[6].fld_field) ;

/* Builds return row field null indicators */
memcpy((void *) city_ind,
 &(fields[0].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;
memcpy((void *) temp_in_f_ind,
 &(fields[1].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;
memcpy((void *) humidity_ind,
 &(fields[2].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;
memcpy((void *) wind_ind,
 &(fields[3].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;
memcpy((void *) wind_velocity_ind,
 &(fields[4].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;
memcpy((void *) barometer_ind,
 &(fields[5].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;
memcpy((void *) forecast_ind,
 &(fields[6].fld_ind),
 sizeof(SQLUDF_NULLIND)) ;

/* Next city weather data */

```

```

 save_area->file_pos++ ;

 break ;

/* Special last call UDF for cleanup (no real args!): Close table */
case SQL_TF_CLOSE:
 /* If you use a weather data text file */
 /* fclose(save_area->file_ptr); */
 /* save_area->file_ptr = NULL; */
 save_area->file_pos = 0 ;
 break ;

}

}

```

Referring to this UDF code, observe that:

- The scratchpad is defined. The row variable is initialized on the OPEN call, and the iptr array and nbr\_rows variable are filled in by the *mystery* function at OPEN time.
- FETCH traverses the iptr array, using row as an index, and moves the values of interest from the current element of iptr to the location pointed to by out\_c1, out\_c2, and out\_c3 result value pointers.
- Finally CLOSE frees the storage acquired by OPEN and anchored in the scratchpad.

Following is the CREATE FUNCTION statement for this UDF:

```

CREATE FUNCTION tfweather_u()
 RETURNS TABLE (CITY VARCHAR(25),
 TEMP_IN_F INTEGER,
 HUMIDITY INTEGER,
 WIND VARCHAR(5),
 WIND_VELOCITY INTEGER,
 BAROMETER FLOAT,
 FORECAST VARCHAR(25))
 SPECIFIC tfweather_u
 DISALLOW PARALLELISM
 NOT FENCED
 DETERMINISTIC
 NO SQL
 NO EXTERNAL ACTION
 SCRATCHPAD
 NO FINAL CALL
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 EXTERNAL NAME 'tf_dml!weather';

```

The above CREATE FUNCTION statement is for a UNIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.

Referring to this statement, observe that:

- It does not take any input, and returns 7 output columns.
- SCRATCHPAD is specified, so DB2 allocates, properly initializes and passes the scratchpad argument.
- NO FINAL CALL is specified.
- The function is specified as NOT DETERMINISTIC, because it depends on more than the SQL input arguments. That is, it depends on the mystery function and we assume that the content can vary from execution to execution.
- DISALLOW PARALLELISM is required for table functions.
- CARDINALITY 100 is an estimate of the expected number of rows returned, provided to the DB2 optimizer.
- DBINFO is not used, and the optimization to only return the columns needed by the particular statement referencing the function is not implemented.
- NOT NULL CALL is specified, so the UDF will not be called if any of its input SQL arguments are NULL, and does not have to check for this condition.

### Example: Function using LOB locators

This UDF takes a locator for an input LOB, and returns a locator for another LOB which is a subset of the input LOB. There are some criteria passed as a second input value, which tell the UDF how exactly to break up the input LOB.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sql.h>
#include <sqlca.h>
#include <sqllda.h>
#include <sqludf.h>
#include "util.h"

void SQL_API_FN lob_subsetter(
 udf_locator * lob_input, /* locator of LOB value to carve up */
 char * criteria, /* criteria for carving */
 udf_locator * lob_output, /* locator of result LOB value */
 short * inp_nul,
 short * cri_nul,
 short * out_nul,
 char * sqlstate,
 char * funcname,
 char * specname,
 char * msgtext) {

 /* local vars */
 short j; /* local indexing var */
```

```

int rc; /* return code variable for API calls */
long input_len; /* receiver for input LOB length */
long input_pos; /* current position for scanning input LOB */
char lob_buf[100]; /* data buffer */
long input_rec; /* number of bytes read by sqludf_substr */
long output_rec; /* number of bytes written by sqludf_append */

/*-----
 * UDF Program Logic Starts Here
 *-----
 * What we do is create an output handle, and then
 * loop over the input, 100 bytes at a time.
 * Depending on the "criteria" passed in, we may decide
 * to append the 100 byte input lob segment to the output, or not.
 *-----
 * Create the output locator, right in the return buffer.
 */

rc = sqludf_create_locator(SQL_TYP_CLOB, &lob_output);
/* Error and exit if unable to create locator */
if (rc) {
 memcpy (sqlstate, "38901", 5);
 /* special sqlstate for this condition */
 goto exit;
}
/* Find out the size of the input LOB value */
rc = sqludf_length(lob_input, &input_len) ;
/* Error and exit if unable to find out length */
if (rc) {
 memcpy (sqlstate, "38902", 5);
 /* special sqlstate for this condition */
 goto exit;
}
/* Loop to read next 100 bytes, and append to result if it meets
 * the criteria.
 */
for (input_pos = 0; (input_pos < input_len); input_pos += 100) {
 /* Read the next 100 (or less) bytes of the input LOB value */
 rc = sqludf_substr(lob_input, input_pos, 100,
 (unsigned char *) lob_buf, &input_rec) ;
 /* Error and exit if unable to read the segment */
 if (rc) {
 memcpy (sqlstate, "38903", 5);
 /* special sqlstate for this condition */
 goto exit;
 }
 /* apply the criteria for appending this segment to result
 * if (...predicate involving buffer and criteria...) {
 * The condition for retaining the segment is TRUE...
 * Write that buffer segment which was last read in
 */
 rc = sqludf_append(lob_output,
 (unsigned char *) lob_buf, input_rec, &output_rec) ;
 /* Error and exit if unable to read the 100 byte segment */
 if (rc) {

```

```

 memcpy (sqlstate, "38904", 5);
 /* special sqlstate for this condition */
 goto exit;
 }
 /* } end if criteria for inclusion met */
} /* end of for loop, processing 100-byte chunks of input LOB
 * if we fall out of for loop, we are successful, and done.
*/
*out_nul = 0;
exit: /* used for errors, which will override null-ness of output. */
 return;
}

```

Referring to this UDF code, observe that:

- There are includes for `sql.h`, where the type `SQL_TYP_CLOB` used in the `sqludf_create_locator()` call is defined, and `sqludf.h`, where the type `udf_locator` is defined.
- The first input argument, and the third input argument (which represents the function output) are defined as pointers to `sqludf_locator`, that is, they represent CREATE FUNCTION specifications of AS LOCATOR.
- There is no testing for nulls of either input argument as NOT NULL CALL is specified in the CREATE FUNCTION statement.
- In the event of error, the UDF exits with `sqlstate` set to 38xxx. This is sufficient to stop the execution of the statement referencing the UDF. The actual 38xxx SQLSTATE values you choose are not important to DB2, but can serve to differentiate the exception conditions which your UDF may encounter.
- The inclusion criteria are left unspecified, but in this case would presumably somehow determine if this particular buffer content passes the test, and presumably would account for the possibility that the last buffer might be a partial buffer.
- By using the `input_rec` variable as the length of the data appended, the UDF takes care of any partial buffer condition.

Following is the CREATE FUNCTION statement for this UDF:

```

CREATE FUNCTION carve(CLOB(50M), VARCHAR(255))
 RETURNS CLOB(50M)
 NOT NULL CALL
 NOT FENCED
 DETERMINISTIC
 NO SQL
 NO EXTERNAL ACTION
 LANGUAGE C
 PARAMETER STYLE DB2SQL
 EXTERNAL NAME '/u/wilfred/udfs/lobudfs!lob_subsetter' ;

```

(This statement is for an AIX version of this UDF. For other platforms, you may need to modify the value specified in the EXTERNAL NAME clause.)



Referring to this statement, observe that:

- NOT NULL CALL is specified, so the UDF will not be called if any of its input SQL arguments are NULL, and does not have to check for this condition.
- The function is defined to be NOT FENCED; recall that the APIs only work in NOT FENCED. NOT FENCED means that the definer will have to have the CREATE\_NOT\_FENCED authority on the database (which is also implied by DBADM authority).
- The function is specified as DETERMINISTIC, meaning that with a given input CLOB value and a given set of criteria, the result will be the same every time.

Now you can successfully run the following statement:

```
UPDATE tablex
 SET col_a = 99,
 col_b = carve (:hv_clob, '...criteria...')
 WHERE tablex_key = :hv_key;
```

The UDF is used to subset the CLOB value represented by the host variable :hv\_clob and update the row represented by key value in host variable :hv\_key.

In this update example by the way, it may be that :hv\_clob is defined in the application as a CLOB\_LOCATOR. It is not this same locator which will be passed to the "carve" UDF! When :hv\_clob is "bound in" to the DB2 engine agent running the statement, it is known only as a CLOB. When it is then passed to the UDF, DB2 generates a new locator for the value. This conversion back and forth between CLOB and locator is not expensive, by the way; it does not involve any extra memory copies or I/O.

### Example: Counter OLE Automation UDF in BASIC

The following example implements a counter class using Microsoft Visual BASIC. The class has an instance variable, nbrOfInvoke, that tracks the number of invocations. The constructor of the class initializes the number to 0. The increment method increments nbrOfInvoke by 1 and returns the current state.

```
Description="Example in SQL Reference"
Name="bert"
Class=bcounter; bcounter.cls
ExeName32="bert_app.exe"
```

```
VERSION 1.0 CLASS
BEGIN
```

```

 SingleUse = -1 'True
END
Attribute VB_Name = "bcounter"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit
Dim nbrOfInvoke As Long

Public Sub increment(output As Long, _
 output_ind As Integer, _
 sqlstate As String, _
 fname As String, _
 fspecname As String, _
 msg As String, _
 scratchpad() As Byte, _
 calltype As Long)

 nbrOfInvoke = nbrOfInvoke + 1

End Sub

Private Sub Class_Initialize()
 nbrOfInvoke = 0
End Sub

Private Sub Class_Terminate()

End Sub

```

The bcounter class is implemented as an OLE automation object and registered under the progId bert.bcounter. You can compile the automation server either as an in-process or local server; this is transparent to DB2. The following CREATE FUNCTION statement registers a UDF bcounter with the increment method as an external implementation:

```

CREATE FUNCTION bcounter () RETURNS integer
EXTERNAL NAME 'bert.bcounter!increment'
LANGUAGE OLE
FENCED
SCRATCHPAD
FINAL CALL
NOT DETERMINISTIC
NULL CALL
PARAMETER STYLE DB2SQL
NO SQL
NO EXTERNAL ACTION
DISALLOW PARALLEL;

```

For the following query:

```
SELECT INT1, BCOUNTER() AS COUNT, INT1/BCOUNTER() AS DIV FROM TEST
```

The results are exactly the same as in the previous example:

| INT1 | COUNT | DIV |    |
|------|-------|-----|----|
|      | 16    | 1   | 16 |
|      | 8     | 2   | 4  |
|      | 4     | 3   | 1  |
|      | 2     | 4   | 0  |
|      | 97    | 5   | 19 |

5 record(s) selected.

### Example: Counter OLE Automation UDF in C++

The following example implements the previous BASIC counter class in C++. Only fragments of the code are shown here, a listing of the entire sample can be found in the /sqllib/samples/ole directory.

The increment method is described in the Object Description Language as part of the counter interface description:

```
interface ICounter : IDispatch
{
 ...
 HRESULT increment([out] long *out,
 [out] short *outnull,
 [out] BSTR *sqlstate,
 [in] BSTR *fname,
 [in] BSTR *fspecname,
 [out] BSTR *msgtext,
 [in,out] SAFEARRAY (unsigned char) *spad,
 [in] long *calltype);
 ...
}
```

The COM CCounter class definition in C++ includes the declaration of the increment method as well as nbrOfInvoke:

```
class FAR CCounter : public ICounter
{
 ...
 STDMETHODIMP CCounter::increment(long *out,
 short *outnull,
 BSTR *sqlstate,
 BSTR *fname,
 BSTR *fspecname,
 BSTR *msgtext,
 SAFEARRAY **spad,
 long *calltype);

 long nbrOfInvoke;
 ...
};
```

The C++ implementation of the method is similar to the BASIC code:

```

STDMETHODIMP CCounter::increment(long *out,
 short *outnull,
 BSTR *sqlstate,
 BSTR *fname,
 BSTR *fspecname,
 BSTR *msgtext,
 SAFEARRAY **spad,
 long *calltype)
{
 nbrOfInvoke = nbrOfInvoke + 1;
 *out = nbrOfInvoke;

 return NOERROR;
};

```

In the above example, `sqlstate` and `msgtext` are [out] parameters of type `BSTR*`, that is, DB2 passes a pointer to NULL to the UDF. To return values for these parameters, the UDF allocates a string and returns it to DB2 (for example, `*sqlstate = SysAllocString (L"01H00")`), and DB2 frees the memory. The parameters `fname` and `fspecname` are [in] parameters. DB2 allocates the memory and passes in values which are read by the UDF, and then DB2 frees the memory.

The class factory of the `CCounter` class creates counter objects. You can register the class factory as a single-use or multi-use object (not shown in this example).

```

STDMETHODIMP CCounterCF::CreateInstance(IUnknown FAR* punkOuter,
 REFIID riid,
 void FAR* FAR* ppv)
{
 CCounter *pObj;
 ...
 // create a new counter object
 pObj = new CCounter;
 ...
};

```

The `CCounter` class is implemented as a local server, and it is registered under the progId `bert.ccounter`. The following `CREATE FUNCTION` statement registers a UDF `ccounter` with the `increment` method as an external implementation:

```

CREATE FUNCTION ccounter () RETURNS integer
EXTERNAL NAME 'bert.ccounter!increment'
LANGUAGE OLE
FENCED
SCRATCHPAD
FINAL CALL
NOT DETERMINISTIC
NULL CALL

```

```

PARAMETER STYLE DB2SQL
NO SQL
NO EXTERNAL ACTION
DISALLOW PARALLEL;

```

While processing the following query, DB2 creates two different instances of class CCounter. An instance is created for each UDF reference in the query. The two instances are reused for the entire query as the scratchpad option is specified in the ccounter UDF registration.

```
SELECT INT1, CCOUNTER() AS COUNT, INT1/CCOUNTER() AS DIV FROM TEST
```

The results are exactly the same as in the previous example:

| INT1 | COUNT | DIV |
|------|-------|-----|
| 16   | 1     | 16  |
| 8    | 2     | 4   |
| 4    | 3     | 1   |
| 2    | 4     | 0   |
| 97   | 5     | 19  |

5 record(s) selected.

### Example: Mail OLE Automation Table Function in BASIC

The following example implements a class using Microsoft Visual BASIC that exposes a public method list to retrieve message header information and the partial message text of messages in Microsoft Exchange. The method implementation employs OLE Messaging which provides an OLE automation interface to MAPI (Messaging API).

```

Description="Mail OLE Automation Table Function"
Module=MainModule; MainModule.bas
Class=Header; Header.cls
ExeName32="tfmapi.dll"
Name="TFMAIL"

```

```

VERSION 1.0 CLASS
BEGIN
 MultiUse = -1 'True
END
Attribute VB_Name = "Header"
Attribute VB_Creatable = True
Attribute VB_Exposed = True
Option Explicit

```

```

Dim MySession As Object
Dim MyMsgColl As Object
Dim MyMsg As Object
Dim CurrentSender As Object
Dim name As Variant
Const SQL_TF_OPEN = -1

```

```

Const SQL_TF_CLOSE = 1
Const SQL_TF_FETCH = 0

Public Sub List(timereceived As Date, subject As String, size As Long, _
 text As String, ind1 As Integer, ind2 As Integer, _
 ind3 As Integer, ind4 As Integer, sqlstate As String, _
 fname As String, fspecname As String, msg As String, _
 scratchpad() As Byte, calltype As Long)

 If (calltype = SQL_TF_OPEN) Then

 Set MySession = CreateObject("MAPI.Session")

 MySession.Logon ProfileName:="Profile1"
 Set MyMsgColl = MySession.Inbox.Messages

 Set MyMsg = MyMsgColl.GetFirst

 ElseIf (calltype = SQL_TF_CLOSE) Then

 MySession.Logoff
 Set MySession = Nothing

 Else

 If (MyMsg Is Nothing) Then

 sqlstate = "02000"

 Else

 timereceived = MyMsg.timereceived
 subject = Left(MyMsg.subject, 15)
 size = MyMsg.size
 text = Left(MyMsg.text, 30)

 Set MyMsg = MyMsgColl.GetNext

 End If
 End If
End Sub

```

On the table function OPEN call, the CreateObject statement creates a mail session, and the logon method logs on to the mail system (user name and password issues are neglected). The message collection of the mail inbox is used to retrieve the first message. On the FETCH calls, the message header information and the first 30 characters of the current message are assigned to the table function output parameters. If no messages are left, SQLSTATE 02000 is returned. On the CLOSE call, the example logs off and sets the session

object to nothing, which releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION MAIL()
 RETURNS TABLE (TIMERECEIVED DATE,
 SUBJECT VARCHAR(15),
 SIZE INTEGER,
 TEXT VARCHAR(30))
 EXTERNAL NAME 'tfmail.header!list'
 LANGUAGE OLE
 PARAMETER STYLE DB2SQL
 NOT DETERMINISTIC
 FENCED
 NULL CALL
 SCRATCHPAD
 FINAL CALL
 NO SQL
 EXTERNAL ACTION
 DISALLOW PARALLEL;
```

Following is a sample query:

```
SELECT * FROM TABLE (MAIL()) AS M
```

| TIMERECEIVED | SUBJECT         | SIZE | TEXT                           |
|--------------|-----------------|------|--------------------------------|
| 01/18/1997   | Welcome!        | 3277 | Welcome to Windows Messaging!  |
| 01/18/1997   | Invoice         | 1382 | Please process this invoice. T |
| 01/19/1997   | Congratulations | 1394 | Congratulations to the purchas |

3 record(s) selected.

---

## Debugging your UDF

It is important to debug your UDF in an environment where you cannot harm the database. You should do your testing on a test database instance until you are absolutely sure your UDF is correct. This is especially true if you plan to define your UDF as NOT FENCED, as a problematic NOT FENCED UDF can bring down an entire instance and this affects all the users. This risk is minimized by using a test instance for debugging. This is because DB2 may not fail gracefully in the case of certain actions by NOT FENCED UDFs, for example, an erroneous modification of storage.

DB2 does check for certain types of limited actions that erroneously modify storage (for example, if the UDF moves a few too many characters to a scratchpad or to the result buffer). In that case, DB2 returns an error, SQLCODE -450 (SQLSTATE 39501), if it detects such a malfunction. DB2 is also designed to fail gracefully in the event of an abnormal termination of a

UDF with SQLCODE -430 (SQLSTATE 38503), or a user interrupt of the UDF's processing with SQLCODE -431 (SQLSTATE 38504).

There is no known scenario where a FENCED UDF can harm an instance. That is why FENCED is the default, and why DB2 demands a special authority to define a UDF as NOT FENCED.

For security and database integrity reasons, it is important to protect the body of your UDF, once it is debugged and defined to DB2. This is particularly true if your UDF is defined as NOT FENCED. If either by accident or with malicious intent, anyone (including yourself) overwrites an operational UDF with code that is not debugged, the UDF could conceivably destroy the database if it is NOT FENCED, or compromise security.

Unfortunately, there is no easy way to run a source-level debugger on a UDF. There are several reasons for this:

- The timing makes it difficult to start the debugger at a time when the UDF is in storage and available
- The UDF runs in a database process with a special UID and the user is not permitted to attach to this process.

Note that valuable debugging tools such as `printf()` do not normally work as debugging aids for your UDF, because the UDF normally runs in a background process where `stdout` has no meaning. As an alternative to using `printf()`, it may be possible for you to instrument your UDF with file output logic, and for debugging purposes write indicative data and control information to a file.

Another technique to debug your UDF is to write a driver program for invoking the UDF outside the database environment. With this technique, you can invoke the UDF with all kinds of marginal or erroneous input arguments to attempt to provoke it into misbehaving. In this environment, it is not a problem to use `printf()` or a source level debugger.



---

# Chapter 12. Using Triggers in an Active DBMS

|                                          |     |                                           |     |
|------------------------------------------|-----|-------------------------------------------|-----|
| Why Use Triggers? . . . . .              | 373 | Functions Within SQL Triggered            |     |
| Benefits and Value of Triggers . . . . . | 374 | Statement . . . . .                       | 383 |
| Overview of a Trigger . . . . .          | 375 | Trigger Cascading . . . . .               | 385 |
| Trigger Event . . . . .                  | 376 | Interactions with Referential Constraints | 385 |
| Set of Affected Rows . . . . .           | 377 | Ordering of Multiple Triggers . . . . .   | 385 |
| Trigger Granularity . . . . .            | 377 | Synergy Between Triggers, Constraints,    |     |
| Trigger Activation Time . . . . .        | 378 | UDTs, UDFs, and LOBs . . . . .            | 386 |
| Transition Variables . . . . .           | 379 | Extracting Information . . . . .          | 386 |
| Transition Tables . . . . .              | 380 | Preventing Operations on Tables . . . . . | 387 |
| Triggered Action . . . . .               | 382 | Defining Business Rules . . . . .         | 388 |
| Triggered Action Condition . . . . .     | 383 | Defining Actions . . . . .                | 388 |
| Triggered SQL Statements . . . . .       | 383 |                                           |     |

---

## Why Use Triggers?

In order to change your database manager from a passive system to an active one, use the capabilities embodied in a trigger function. A *trigger* defines a set of actions that are activated or *triggered* by an update operation on a specified base table. These actions may cause other changes to the database, perform operations outside DB2 (for example, send an e-mail or write a record in a file), raise an exception to prevent the update operation from taking place, and so on.

You can use triggers to support general forms of integrity such as business rules. For example, your business may wish to refuse orders that exceed its customers' credit limit. A trigger can be used to enforce this constraint. In general, triggers are powerful mechanisms to capture *transitional* business rules. Transitional business rules are rules that involve different states of the data.

For example, suppose a salary cannot be increased by more than 10 per cent. To check this rule, the value of the salary before and after the increase must be compared. For rules that do not involve more than one state of the data, check and referential integrity constraints may be more appropriate (refer to the *SQL Reference* for more information). Because of the declarative semantics of check and referential constraints, their use is recommended for constraints that are not transitional.

You can also use triggers for tasks such as automatically updating summary data. By keeping these actions as a part of the database and ensuring that

they occur automatically, triggers enhance database integrity. For example, suppose you want to automatically track the number of employees managed by a company:

```
Tables: EMPLOYEE (as in Sample Tables)
 COMPANY_STATS (NBEMP, NBPRODUCT, REVENUE)
```

You can define two triggers:

- A trigger that increments the number of employees each time a new person is hired, that is, each time a new row is inserted into the table EMPLOYEE:

```
CREATE TRIGGER NEW_HIRED
AFTER INSERT ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

- A trigger that decrements the number of employees each time an employee leaves the company, that is, each time a row is deleted from the table EMPLOYEE:

```
CREATE TRIGGER FORMER_EMP
AFTER DELETE ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

Specifically, you can use triggers to:

- Validate input data using the SIGNAL SQLSTATE SQL statement, the built-in RAISE\_ERROR function, or invoke a UDF to return an SQLSTATE indicating that an error has occurred, if invalid data is discovered. Note that validation of non-transitional data is usually better handled by check and referential constraints. By contrast, triggers are appropriate for validation of transitional data, that is, validations which require comparisons between the value before and after an update operation.
- Automatically generate values for newly inserted rows (this is known as a *surrogate function*). That is, to implement user-defined default values, possibly based on other values in the row or values in other tables.
- Read from other tables for cross-referencing purposes.
- Write to other tables for audit-trail purposes.
- Support *alerts* (for example, through electronic mail messages).

## Benefits and Value of Triggers

Using triggers in your database manager can result in:

- **Faster application development.**  
Because triggers are stored in the relational database, the actions performed by triggers do not have to be coded in each application.
- **Global enforcement of business rules**

A trigger only has to be defined once, and then it can be used for any application that changes the table.

- **Easier maintenance**

If a business policy changes, only the corresponding trigger needs to change instead of each application program.

---

## Overview of a Trigger

When you create a trigger, you associate it with a table. This table is called the *subject table* of the trigger. The term *update operation* refers to any change in the state of the subject table. An update operation is initiated by:

- an INSERT statement
- an UPDATE statement, or a referential constraint which performs an UPDATE
- a DELETE statement, or a referential constraint which performs a DELETE

You must associate each trigger with one of these three types of update operations. The association is called the *trigger event* for that particular trigger.

You must also define the action, called the *triggered action*, that the trigger performs when its trigger event occurs. The triggered action consists of one or more SQL statements which can execute either before or after the database manager performs the trigger event. Once a trigger event occurs, the database manager determines the set of rows in the subject table that the update operation affects and executes the trigger.

When you create a trigger, you declare the following attributes and behavior:

- The name of the trigger.
- The name of the subject table.
- The trigger activation time (BEFORE or AFTER the update operation executes).
- The trigger event (INSERT, DELETE, or UPDATE).
- The old values transition variable, if any.
- The new values transition variable, if any.
- The old values transition table, if any.
- The new values transition table, if any.
- The granularity (FOR EACH STATEMENT or FOR EACH ROW).
- The triggered action of the trigger (including a triggered action condition and triggered SQL statement(s)).
- If the trigger event is UPDATE, then the trigger column list for the trigger event of the trigger, as well as an indication of whether the trigger column list was explicit or implicit.

- The trigger creation timestamp.
- The current function path.

For more information on the CREATE TRIGGER statement, refer to the *SQL Reference*.

---

## Trigger Event

Every trigger is associated with an event. Triggers are activated when their corresponding event occurs in the database. This trigger event occurs when the specified action, either an UPDATE, INSERT, or DELETE (including those caused by actions of referential constraints), is performed on the subject table. For example:

```
CREATE TRIGGER NEW_HIRE
 AFTER INSERT ON EMPLOYEE
 FOR EACH ROW MODE DB2SQL
 UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

The above statement defines the trigger `new_hire`, which activates when you perform an insert operation on table `employee`.

You associate every trigger event, and consequently every trigger, with exactly one subject table and exactly one update operation. The update operations are:

### Insert operation

An insert operation can only be caused by an INSERT statement. Therefore, triggers are not activated when data is loaded using utilities that do not use INSERT, such as the LOAD command.

### Update operation

An update operation can be caused by an UPDATE statement or as a result of a referential constraint rule of ON DELETE SET NULL.

### Delete operation

A delete operation can be caused by a DELETE statement or as a result of a referential constraint rule of ON DELETE CASCADE.

If the trigger event is an update operation, the event can be associated with specific columns of the subject table. In this case, the trigger is only activated if the update operation attempts to update any of the specified columns. This provides a further refinement of the event that activates the trigger. For example, the following trigger, `REORDER`, activates only if you perform an update operation on the columns `ON_HAND` or `MAX_STOCKED`, of the table `PARTS`.

```
CREATE TRIGGER REORDER
 AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
 REFERENCING NEW AS N_ROW
 FOR EACH ROW MODE DB2SQL
```

```

WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
BEGIN ATOMIC
VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
 N_ROW.ON_HAND,
 N_ROW.PARTNO));
END

```

---

## Set of Affected Rows

A trigger event defines a set of rows in the subject table that is affected by that SQL operation. For example, suppose you run the following UPDATE statement on the parts table:

```

UPDATE PARTS
SET ON_HAND = ON_HAND + 100
WHERE PART_NO > 15000

```

The set of affected rows for the associated trigger contains all the rows in the parts table whose *part\_no* is greater than 15 000.

---

## Trigger Granularity

When a trigger is activated, it runs according to its granularity as follows:

### **FOR EACH ROW**

It runs as many times as the number of rows in the set of affected rows.

### **FOR EACH STATEMENT**

It runs once for the entire trigger event.

If the set of affected rows is empty (that is, in the case of a searched UPDATE or DELETE in which the WHERE clause did not qualify any rows), a FOR EACH ROW trigger does not run. But a FOR EACH STATEMENT trigger still runs once.

For example, keeping a count of number of employees can be done using FOR EACH ROW.

```

CREATE TRIGGER NEW_HIRED
AFTER INSERT ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1

```

You can achieve the same affect with one update by using a granularity of FOR EACH STATEMENT.

```

CREATE TRIGGER NEW_HIRED
AFTER INSERT ON EMPLOYEE
REFERENCING NEW_TABLE AS NEWEMPS

```

```
FOR EACH STATEMENT MODE DB2SQL
UPDATE COMPANY_STATS
SET NBEMP = NBEMP + (SELECT COUNT(*) FROM NEWEMPS)
```

**Note:** A granularity of FOR EACH STATEMENT is not supported for BEFORE triggers (discussed in “Trigger Activation Time”).

---

## Trigger Activation Time

The *trigger activation time* specifies when the trigger should be activated. That is, either BEFORE or AFTER the trigger event executes. For example, the activation time of the following trigger is AFTER the INSERT operation on employee.

```
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

If the activation time is BEFORE, the triggered actions are activated for each row in the set of affected rows before the trigger event executes. Note that BEFORE triggers must have a granularity of FOR EACH ROW.

If the activation time is AFTER, the triggered actions are activated for each row in the set of affected rows or for the statement, depending on the trigger granularity. This occurs after the trigger event executes, and after the database manager checks all constraints that the trigger event may affect, including actions of referential constraints. Note that AFTER triggers can have a granularity of either FOR EACH ROW or FOR EACH STATEMENT.

The different activation times of triggers reflect different purposes of triggers. Basically, BEFORE triggers are an extension to the constraint subsystem of the database management system. Therefore, you generally use them to:

- Perform validation of input data,
- Automatically generate values for newly inserted rows
- Read from other tables for cross-referencing purposes.

BEFORE triggers are not used for further modifying the database because they are activated before the trigger event is applied to the database. Consequently, they are activated before integrity constraints are checked and may be violated by the trigger event.

Conversely, you can view AFTER triggers as a module of application logic that runs in the database every time a specific event occurs. As a part of an application, AFTER triggers always see the database in a consistent state. Note that they are run after the integrity constraints that may be violated by the

triggering SQL operation have been checked. Consequently, you can use them mostly to perform operations that an application can also perform. For example:

- Perform follow on update operations in the database
- Perform actions outside the database, for example, to support alerts. Note that actions performed outside the database are not rolled back if the trigger is rolled back.

Because of the different nature of BEFORE and AFTER triggers, a different set of SQL operations can be used to define the triggered actions of BEFORE and AFTER triggers. For example, update operations are not allowed in BEFORE triggers because there is no guarantee that integrity constraints will not be violated by the triggered action. The set of SQL operations you can specify in BEFORE and AFTER triggers are described in “Triggered Action” on page 382. Similarly, different trigger granularities are supported in BEFORE and AFTER triggers. For example, FOR EACH STATEMENT is not allowed in BEFORE triggers because there is no guarantee that constraints will not be violated by the triggered action, which would, in turn, result in failure of the operation.

---

## Transition Variables

When you carry out a FOR EACH ROW trigger, it may be necessary to refer to the value of columns of the row in the set of affected rows, for which the trigger is currently executing. Note that to refer to columns in tables in the database (including the subject table), you can use regular SELECT statements. A FOR EACH ROW trigger may refer to the columns of the row for which it is currently executing by using two transition variables that you can specify in the REFERENCING clause of a CREATE TRIGGER statement. There are two kinds of transition variables, which are specified as *OLD* and *NEW*, together with a correlation-name. They have the following semantics:

### **OLD correlation-name**

Specifies a correlation name which captures the original state of the row, that is, before the triggered action is applied to the database.

### **NEW correlation-name**

Specifies a correlation name which captures the value that is, or was, used to update the row in the database when the triggered action is applied to the database.

Consider the following example:

```
CREATE TRIGGER REORDER
AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
REFERENCING NEW AS N_ROW
FOR EACH ROW MODE DB2SQL
WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED
```

```

AND N_ROW.ORDER_PENDING = 'N')
BEGIN ATOMIC
 VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
 N_ROW.ON_HAND,
 N_ROW.PARTNO));
 UPDATE PARTS SET PARTS.ORDER_PENDING = 'Y'
 WHERE PARTS.PARTNO = N_ROW.PARTNO;
END

```

Based on the definition of the OLD and NEW transition variables given above, it is clear that not every transition variable can be defined for every trigger. Transition variables can be defined depending on the kind of trigger event:

#### **UPDATE**

An UPDATE trigger can refer to both OLD and NEW transition variables.

#### **INSERT**

An INSERT trigger can only refer to a NEW transition variable because before the activation of the INSERT operation, the affected row does not exist in the database. That is, there is no original state of the row that would define old values before the triggered action is applied to the database.

#### **DELETE**

A DELETE trigger can only refer to an OLD transition variable because there are no new values specified in the delete operation.

**Note:** Transition variables can only be specified for FOR EACH ROW triggers. In a FOR EACH STATEMENT trigger, a reference to a transition variable is not sufficient to specify to which of the several rows in the set of affected rows the transition variable is referring.

---

## **Transition Tables**

In both FOR EACH ROW and FOR EACH STATEMENT triggers, it may be necessary to refer to the whole set of affected rows. This is necessary, for example, if the trigger body needs to apply aggregations over the set of affected rows (for example, MAX, MIN, or AVG of some column values). A trigger may refer to the set of affected rows by using two transition tables that can be specified in the REFERENCING clause of a CREATE TRIGGER statement. Just like the transition variables, there are two kinds of transition tables, which are specified as OLD\_TABLE and NEW\_TABLE together with a *table-name*, with the following semantics:



**OLD\_TABLE table-name**

Specifies the name of the table which captures the original state of the set of affected rows (that is, before the triggering SQL operation is applied to the database).

**NEW\_TABLE table-name**

Specifies the name of the table which captures the value that is used to update the rows in the database when the triggered action is applied to the database.

For example:

```
CREATE TRIGGER REORDER
AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
REFERENCING NEW_TABLE AS N_TABLE
NEW AS N_ROW
FOR EACH ROW MODE DB2SQL
WHEN ((SELECT AVG (ON_HAND) FROM N_TABLE) > 35)
BEGIN ATOMIC
 VALUES(INFORM_SUPERVISOR(N_ROW.PARTNO,
 N_ROW.MAX_STOCKED,
 N_ROW.ON_HAND));
END
```

Note that NEW\_TABLE always has the full set of updated rows, even on a FOR EACH ROW trigger. When a trigger acts on the table on which the trigger is defined, NEW\_TABLE contains the changed rows from the statement that activated the trigger. However, NEW\_TABLE does not contain the changed rows that were caused by statements within the trigger, as that would cause a separate activation of the trigger.

The transition tables are read-only. The same rules that define the kinds of transition variables that can be defined for which trigger event, apply for transition tables:

**UPDATE**

An UPDATE trigger can refer to both OLD\_TABLE and NEW\_TABLE transition tables.

**INSERT**

An INSERT trigger can only refer to a NEW\_TABLE transition table because before the activation of the INSERT operation the affected rows do not exist in the database. That is, there is no *original state of the rows* that defines old values before the triggered action is applied to the database.

**DELETE**

A DELETE trigger can only refer to an OLD transition table because there are no new values specified in the delete operation.

**Note:** It is important to observe that transition tables can be specified for both granularities of AFTER triggers: FOR EACH ROW and FOR EACH STATEMENT.

The scope of the OLD\_TABLE and NEW\_TABLE *table-name* is the trigger body. In this scope, this name takes precedence over the name of any other table with the same unqualified *table-name* that may exist in the schema. Therefore, if the OLD\_TABLE or NEW\_TABLE *table-name* is for example, X, a reference to X (that is, an unqualified X) in the FROM clause of a SELECT statement will always refer to the transition table even if there is a table named X in the in the schema of the trigger creator. In this case, the user has to make use of the fully qualified name in order to refer to the table X in the schema.

---

## Triggered Action

The activation of a trigger results in the running of its associated triggered action. Every trigger has exactly one triggered action which, in turn, has two components:

- An optional *triggered action condition* or WHEN clause
- A set of *triggered SQL statement(s)*.

The triggered action condition defines whether or not the set of triggered statements are performed for the row or for the statement for which the triggered action is executing. The set of triggered statements define the set of actions performed by the trigger in the database as a consequence of its event having occurred.

For example, the following trigger action specifies that the set of triggered SQL statements should only be activated for rows in which the value of the on\_hand column is less than ten per cent of the value of the max\_stocked column. In this case, the set of triggered SQL statements is the invocation of the issue\_ship\_request function.

```
CREATE TRIGGER REORDER
 AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
 REFERENCING NEW AS N_ROW
 FOR EACH ROW MODE DB2SQL

 WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
 BEGIN ATOMIC
 VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
 N_ROW.ON_HAND,
 N_ROW.PARTNO));
 END
```

## Triggered Action Condition

As explained in “Triggered Action” on page 382, the *triggered action condition* is an optional clause of the triggered action which specifies a search condition that must evaluate to *true* to run SQL statements within the triggered action. If the WHEN clause is omitted, then the SQL statements within the triggered action are always executed.

The triggered action condition is evaluated once for each row if the trigger is a FOR EACH ROW trigger, and once for the statement if the trigger is a FOR EACH STATEMENT trigger.

This clause provides further control that you can use to fine tune the actions activated on behalf of a trigger. An example of the usefulness of the WHEN clause is to enforce a data dependent rule in which a triggered action is activated only if the incoming value falls inside or outside of a certain range.

## Triggered SQL Statements

The set of triggered SQL statements carries out the *real* actions caused by activating a trigger. As described previously, not every SQL operation is meaningful in every trigger. Depending on whether the trigger activation time is BEFORE or AFTER, different kinds of operations may be appropriate as a triggered SQL statement.

For a list of triggered SQL statements, and additional information on BEFORE and AFTER triggers, refer to the *SQL Reference*.

In most cases, if any triggered SQL statement returns a negative return code, the triggering SQL statement together with all trigger and referential constraint actions are rolled back, and an error is returned: SQLCODE -723 (SQLSTATE 09000). The trigger name, SQLCODE, SQLSTATE and many of the tokens from the failing triggered SQL statement are returned. Error conditions occurring when triggers are running that are critical or roll back the entire unit of work are not returned using SQLCODE -723 (SQLSTATE 09000).

## Functions Within SQL Triggered Statement

Functions, including user-defined functions (UDFs), may be invoked within a triggered SQL statement. Consider the following example:

```
CREATE TRIGGER REORDER
 AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
 REFERENCING NEW AS N_ROW
 FOR EACH ROW MODE DB2SQL
 WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
```

```

BEGIN ATOMIC
 VALUES (ISSUE_SHIP_REQUEST (N_ROW.MAX_STOCKED - N_ROW.ON_HAND,
 N_ROW.PARTNO));
END

```

When a triggered SQL statement contains a function invocation with an unqualified function name, the function invocation is resolved based on the function path at the time of creation of the trigger. For details on the resolution of functions, refer to the *SQL Reference*.

UDFs are written in either the C or C++ programming language. This enables control of logic flows, error handling and recovery, and access to system and library functions. (See “Chapter 11. Writing User-Defined Functions (UDFs)” on page 287 for a description of UDFs.) This capability allows a triggered action to perform non-SQL types of operations when a trigger is activated. For example, such a UDF could send an electronic mail message and thereby act as an alert mechanism. External actions, such as messages, are not under commit control and will be run regardless of success or failure of the rest of the triggered actions.

Also, the function can return an SQLSTATE that indicates an error has occurred which results in the failure of the triggering SQL statement. This is one method of implementing user-defined constraints. (Using a SIGNAL SQLSTATE statement is the other.) In order to use a trigger as a means to check complex user-defined constraints, you can use the RAISE\_ERROR built-in function in a triggered SQL statement. This function can be used to return a user-defined SQLSTATE (SQLCODE -438) to applications. For details on invocation and use of this function, refer to the *SQL Reference*.

For example, consider some rules related to the HIREDATE column of the EMPLOYEE table, where HIREDATE is the date that the employee starts working.

- HIREDATE must be date of insert or a future date
- HIREDATE cannot be more than 1 year from date of insert.
- If HIREDATE is between 6 and 12 months from date of insert, notify personnel manager using a UDF called send\_note.

The following trigger handles all of these rules on INSERT:

```

CREATE TRIGGER CHECK_HIREDATE
NO CASCADE BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEW_EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
VALUES CASE
 WHEN NEW_EMP.HIREDATE < CURRENT DATE
 THEN RAISE_ERROR('85001', 'HIREDATE has passed')
 WHEN NEW_EMP.HIREDATE - CURRENT DATE > 10000.

```

```

 THEN RAISE_ERROR('85002', 'HIREDATE too far out')
 WHEN NEW_EMP.HIREDATE - CURRENT DATE > 600.
 THEN SEND_MOTE('persmgr',NEW_EMP.EMPNO,'late.txt')
 END;
END

```

---

## Trigger Cascading

When you run a triggered SQL statement, it may cause the event of another, or even the same, trigger to occur, which in turn, causes the other, (or a second instance of the same) trigger to be activated. Therefore, activating a trigger can cascade the activation of one or more other triggers.

The run-time depth level of trigger cascading supported is 16. If a trigger at level 17 is activated, SQLCODE -724 (SQLSTATE 54038) will be returned and the triggering statement will be rolled back.

---

## Interactions with Referential Constraints

As described above, the trigger event can be the result of changes due to referential constraint enforcement. For example, given two tables DEPT and EMP, if deleting or updating DEPT causes propagated deletes or updates to EMP by means of referential integrity constraints, then delete or update triggers defined on EMP become activated as a result of the referential constraint defined on DEPT. The triggers on EMP are run either BEFORE or AFTER the deletion (in the case of ON DELETE CASCADE) or update of rows in EMP (in the case of ON DELETE SET NULL), depending on their activation time.

---

## Ordering of Multiple Triggers

When triggers are defined using the CREATE TRIGGER statement, their creation time is registered in the database in form of a timestamp. The value of this timestamp is subsequently used to order the activation of triggers when there is more than one trigger that should be run at the same time. For example, the timestamp is used when there is more than one trigger on the same subject table with the same event and the same activation time. The timestamp is also used when there is one or more AFTER triggers that are activated by the trigger event and referential constraint actions caused directly or indirectly (that is, recursively by other referential constraints) by the triggered action. Consider the following two triggers:

```

CREATE TRIGGER NEW_HIRED
AFTER INSERT ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC

```

```

UPDATE COMPANY_STATS
SET NBEMP = NBEMP + 1;
END;

CREATE TRIGGER NEW_HIRED_DEPT
AFTER INSERT ON EMPLOYEE
REFERENCING NEW AS EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
 UPDATE DEPTS
 SET NBEMP = NBEMP + 1
 WHERE DEPT_ID = EMP.DEPT_ID;
END;

```

The above triggers are activated when you run an INSERT operation on the employee table. In this case, the timestamp of their creation defines which of the above two triggers is activated first.

The activation of the triggers is conducted in ascending order of the timestamp value. Thus, a trigger that is newly added to a database runs after all the other triggers that are previously defined.

Old triggers are activated before new triggers to ensure that new triggers can be used as *incremental* additions to the changes that affect the database. For example, if a triggered SQL statement of trigger T1 inserts a new row into a table T, a triggered SQL statement of trigger T2 that is run after T1 can be used to update the same row in T with specific values. By activating triggers in ascending order of creation, you can ensure that the actions of new triggers run on a database that reflects the result of the activation of all old triggers.

---

## Synergy Between Triggers, Constraints, UDTs, UDFs, and LOBs

“Combining UDTs, UDFs, and LOBs” on page 223 describes how to combine UDTs, UDFs, and LOBs to represent and manipulate large, complex structures. The following section describes how to exploit triggers and constraints to complete the modeling of such application structures. With triggers, you can:

- Extract information from these structures to keep them explicitly in columns of tables (instead of hidden within the structure)
- Define the integrity rules that govern these structures in the application domain
- Express important actions that need to be taken under certain values of the structures.

### Extracting Information

In “Combining UDTs, UDFs, and LOBs” on page 223, the complete electronic mail is stored within the column message of the ELECTRONIC\_MAIL table.

To manipulate the electronic mail, UDFs were used to extract information from the message column every time such information was required within an SQL statement.

Notice that the queries do not extract information once and store it explicitly as columns of tables. If this was done, it would increase the performance of the queries, not only because the UDFs are not invoked repeatedly, but also because you can then define indexes on the extracted information.

Using triggers, you can extract this information whenever new electronic mail is stored in the database. To achieve this, add new columns to the ELECTRONIC\_MAIL table and define a BEFORE trigger to extract the corresponding information as follows:

```
ALTER TABLE ELECTRONIC_MAIL
 ADD COLUMN SENDER VARCHAR (200)
 ADD COLUMN RECEIVER VARCHAR (200)
 ADD COLUMN SENT_ON DATE
 ADD COLUMN SUBJECT VARCHAR (200)

CREATE TRIGGER EXTRACT_INFO
 NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
 REFERENCING NEW AS N
 FOR EACH ROW MODE DB2SQL
 BEGIN ATOMIC
 SET N.SENDER = SENDER(N.MESSAGE);
 SET N.RECEIVER = RECEIVER(N.MESSAGE);
 SET N.SENT_ON = SENDING_DATE(N.MESSAGE);
 SET N.SUBJECT = SUBJECT(N.MESSAGE);
 END
```

Now, whenever new electronic mail is inserted into the message column, its sender, its receiver, the date on which it was sent, and its subject are extracted from the message and stored in separate columns.

## Preventing Operations on Tables

Suppose you want to prevent mail you sent, which was undelivered and returned to you (perhaps because the e-mail address was incorrect), from being stored in the e-mail's table.

To do so, you need to prevent the execution of certain SQL INSERT statements. There are two ways to do this:

- Define a BEFORE trigger that raises an error whenever the subject of an e-mail is *undelivered mail*:

```
CREATE TRIGGER BLOCK_INSERT
 NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
 REFERENCING NEW AS N
 FOR EACH ROW MODE DB2SQL
```

```

WHEN (SUBJECT(N.MESSAGE) = 'undelivered mail')
BEGIN ATOMIC
 SIGNAL SQLSTATE '85101' ('Attempt to insert undelivered mail');
END

```

- Define a check constraint forcing values of the new column subject to be different from *undelivered mail*:

```

ALTER TABLE ELECTRONIC_MAIL
ADD CONSTRAINT NO_UNDELIVERED
CHECK (SUBJECT <> 'undelivered mail')

```

Because of the advantages of the declarative nature of constraints, the constraint should generally be defined instead of the trigger.

## Defining Business Rules

Suppose your company has the policy that all e-mail dealing with customer complaints must have Mr. Nelson, the marketing manager, in the carbon copy (CC) list. Because this is a rule, you might want to express it as a constraint such as one of the following (assuming the existence of a CC\_LIST UDF to check it):

```

ALTER TABLE ELECTRONIC_MAIL ADD
CHECK (SUBJECT <> 'Customer complaint' OR
 CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 1)

```

However, such a constraint prevents the insertion of e-mail dealing with customer complaints that do not have the marketing manager in the cc list. This is certainly not the intent of your company's business rule. The intent is to forward to the marketing manager any e-mail dealing with customer complaints that were not copied to the marketing manager. Such a business rule can only be expressed with a trigger because it requires taking actions that cannot be expressed with declarative constraints. The trigger assumes the existence of a SEND\_NOTE function with parameters of type E\_MAIL and character string.

```

CREATE TRIGGER INFORM_MANAGER
AFTER INSERT ON ELECTRONIC_MAIL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.SUBJECT = 'Customer complaint' AND
 CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 0)
BEGIN ATOMIC
 VALUES(SEND_NOTE(N.MESSAGE, 'nelson@vnet.ibm.com'));
END

```

## Defining Actions

Now assume that your general manager wants to keep the names of customers who have sent three or more complaints in the last 72 hours in a separate table. The general manager also wants to be informed whenever a customer name is inserted in this table more than once.



To define such actions, you define:

- An UNHAPPY\_CUSTOMERS table:

```
CREATE TABLE UNHAPPY_CUSTOMERS (
 NAME VARCHAR (30),
 EMAIL_ADDRESS VARCHAR (200),
 INSERTION_DATE DATE)
```

- A trigger to automatically insert a row in UNHAPPY\_CUSTOMERS if 3 or more messages were received in the last 3 days (assumes the existence of a CUSTOMERS table that includes a NAME column and an E\_MAIL\_ADDRESS column):

```
CREATE TRIGGER STORE_UNHAPPY_CUST
AFTER INSERT ON ELECTRONIC_MAIL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (3 <= (SELECT COUNT(*)
 FROM ELECTRONIC_MAIL
 WHERE SENDER = N.SENDER
 AND SENDING_DATE(MESSAGE) > CURRENT DATE - 3 DAYS)
)
BEGIN ATOMIC
 INSERT INTO UNHAPPY_CUSTOMERS
 VALUES ((SELECT NAME
 FROM CUSTOMERS
 WHERE E_MAIL_ADDRESS = N.SENDER), N.SENDER, CURRENT DATE);
END
```

- A trigger to send a note to the general manager if the same customer is inserted in UNHAPPY\_CUSTOMERS more than once (assumes the existence of a SEND\_NOTE function that takes 2 character strings as input):

```
CREATE TRIGGER INFORM_GEN_MGR
AFTER INSERT ON UNHAPPY_CUSTOMERS
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (1 < (SELECT COUNT(*)
 FROM UNHAPPY_CUSTOMERS
 WHERE EMAIL_ADDRESS = N.EMAIL_ADDRESS))
)
BEGIN ATOMIC
 VALUES(SEND_NOTE('Check customer:' CONCAT N.NAME,
 'bigboss@vnet.ibm.com'));
END
```



---

## Part 5. DB2 Programming Considerations



---

## Chapter 13. Programming in Complex Environments

|                                          |     |                                          |     |
|------------------------------------------|-----|------------------------------------------|-----|
| National Language Support Considerations | 394 | Unequal Code Page Situations             | 425 |
| Collating Sequences                      | 394 | Client-Based Parameter Validation        | 428 |
| Overview                                 | 394 | Using the DESCRIBE Statement             | 429 |
| Collating Sequence Sort Order:           |     | Using Fixed or Variable Length Data      |     |
| EBCDIC and ASCII Example                 | 397 | Types                                    | 430 |
| Specifying a Collating Sequence          | 398 | Character Conversion String Length       |     |
| Deriving Code Page Values                | 401 | Overflow                                 | 431 |
| Deriving Locales in Application          |     | Applications Connected to a Unicode      |     |
| Programs                                 | 402 | (UCS-2) Database                         | 433 |
| How DB2 Derives Locales                  | 402 | Considerations for Multisite Updates     | 433 |
| Programming Considerations               | 403 | Remote Unit of Work                      | 434 |
| Coding SQL Statements                    | 403 | Multisite Update                         | 434 |
| Coding Remote Stored Procedures          |     | When to Use Multisite Update             | 435 |
| and UDFs                                 | 405 | Coding SQL for a Multisite Update        |     |
| Package Name Considerations in           |     | Application                              | 435 |
| Mixed Code Page Environments             | 405 | Precompiling a Multisite Update          |     |
| Precompiling and Binding                 | 406 | Application                              | 437 |
| Executing an Application                 | 406 | Specifying Configuration Parameters      |     |
| A Note of Caution                        | 406 | for a Multisite Update Application       | 439 |
| Conversion Between Different Code        |     | Multisite Update Restrictions            | 440 |
| Pages                                    | 407 | Accessing Host or AS/400 Servers         | 441 |
| Unicode/UCS-2 and UTF-8 Support in       |     | Multiple Thread Database Access          | 441 |
| DB2 UDB                                  | 410 | Recommendations for Using Multiple       |     |
| Introduction                             | 410 | Threads                                  | 443 |
| UCS-2/UTF-8 Implementation in DB2        |     | Multithreaded UNIX Applications          |     |
| UDB                                      | 411 | Working with Code Page and Country       |     |
| Running CLI/ODBC/JDBC/SQLJ               |     | Code                                     | 443 |
| Programs in a DBCS Environment           | 418 | Potential Pitfalls when Using Multiple   |     |
| Japanese and Traditional Chinese EUC and |     | Threads                                  | 444 |
| UCS-2 Code Set Considerations            | 419 | Preventing Deadlocks for Multiple        |     |
| Mixed EUC and Double-Byte Client and     |     | Contexts                                 | 445 |
| Database Considerations                  | 421 | Concurrent Transactions                  | 446 |
| Considerations for Traditional Chinese   |     | Potential Pitfalls when Using Concurrent |     |
| Users                                    | 422 | Transactions                             | 446 |
| Developing Japanese or Traditional       |     | Preventing Deadlocks for Concurrent      |     |
| Chinese EUC Applications                 | 422 | Transactions                             | 447 |
| Graphic Data Handling                    | 423 | X/Open XA Interface Programming          |     |
| Considerations for DBCLOB Files          | 424 | Considerations                           | 448 |
| Collation                                | 424 | Application Linkage                      | 451 |
| Developing for Mixed Code Set            |     | Working with Large Volumes of Data       |     |
| Environments                             | 424 | Across a Network                         | 451 |

---

## National Language Support Considerations

This section describes National Language Support (NLS) support issues that you must consider for your applications. The major topics discussed are:

- Collating Sequences
- Deriving Code Page Values
- Deriving Locales in Application Programs
- Programming Considerations
- Conversion Between Different Code Pages

### Collating Sequences

The database manager compares character data using a *collating sequence*. This is an ordering for a set of characters that determines whether a particular character sorts higher, lower, or the same as another.

**Note:** Character string data defined with the FOR BIT DATA attribute, or BLOB data, is sorted using the binary sort sequence.

For example, a collating sequence can be used to indicate that lowercase and uppercase versions of a particular character are to be sorted equally.

The database manager allows databases to be created with custom collating sequences. The following sections help you determine and implement a particular collating sequence for a database.

#### Overview

In a database, each single-byte character is represented internally as a unique number between 0 and 255, (in hexadecimal notation, between X'00' and X'FF'). This number is referred to as the *code point* of the character; the assignments of numbers to characters in a set are collectively called a **code page**. A collating sequence is a mapping between the code point and the desired position of each character in a sorted sequence. The numeric value of the position is called the *weight* of the character in the collating sequence. The simplest collating sequence is one where the weights are identical to the code points. This is called the *identity sequence*.

For example, consider the characters B (X'42'), and b (X'62'). If, according to the collating sequence table, they both have a sort weight of X'42' (B), then they collate the same. If the sort weight for B is X'9E' and the sort weight for b is X'9D', then b will be sorted before B. Actual weights depend on the collating sequence table used which depends on the code set and locale. Note that a collating sequence table is not the same as a code page table, which defines code points.

Consider the following example. In ASCII, the characters A through Z are represented by X'41' through X'5A'. To describe a collating sequence where these are sorted in order, and consecutively (no intervening characters), you can write X'41', X'42', ...X'59', X'5A'.

For multi-byte characters, the hexadecimal value of the multi-byte character is also used as the weight. For example, X'8260', X'8261' are the code points for double byte character A and B. In this case, the collation weights for X'82', X'60', and X'61' are used to sort these two characters according to their code points.

The values of the weights in a collating sequence need not be unique. For example, you could give uppercase letters and their lowercase equivalents the same weight.

Specifying the collating sequence can be simplified if a collating sequence provides weights for all 256 code points. The weight of each character can be determined using the code point of the character. This is the method used to specify a collating sequence for the database manager: a string of 256 bytes, where the *n*th byte (starting with 0) contains the weight of code point *n*.

In all cases, DB2 uses the collation table which was specified at database creation time. If you require the multi-byte characters to sort the way they appear in their code point table, you must specify IDENTITY as your collation sequence when you create the database.

**Note:** For DBCS characters in GRAPHIC fields, the sort sequence is always IDENTITY without regard to the collation sequence specified at database creation time.

**Character Comparisons:** Once a collating sequence is established, character comparison is performed by comparing the weights of two characters, instead of directly comparing their code point values.

If weights that are not unique are used, characters that are not identical may compare equally. Because of this, string comparison must be a two-phase process:

1. Compare the characters of each string based on their weights.
2. If step 1 yielded equality, compare the characters of each string based on their code point values.

If the collating sequence contains 256 unique weights, only the first step is performed. If the collating sequence is the identity sequence only the second step is performed. In either case, there is a performance benefit.

For more information on character comparisons, see the *SQL Reference*.

**Case Independent Comparisons:** To perform character comparisons that are independent of whether they are upper or lower case, you can use the TRANSLATE function to select and compare mixed case column data by translating it to upper case, but only for the purposes of comparison. Consider the following data:

```
Abe1
abe1s
ABEL
abe1
ab
Ab
```

For the following select statement:

```
SELECT c1 FROM T1 WHERE TRANSLATE(c1) LIKE 'AB%'
```

you would receive the following results:

```
ab
Ab
abe1
Abe1
ABEL
abe1s
```

**Note:** You could also set the select as in the following view v1, and then make all your comparisons against the view (in upper case) and your inserts into the table in mixed case:

```
CREATE VIEW v1 AS SELECT TRANSLATE(c1) FROM t1
```

At the database level, you can set the *collating sequence* as part of the CREATE DATABASE API. This allows you to decide if 'a' is processed before 'A', or if 'A' is processed after 'a', or if they are processed with equal weighting. This will make them equal when collating or sorting using the ORDER BY clause. If you have two values of 'a' and 'A', 'A' will always come before 'a', because in all senses they are equal, so the only difference upon which to sort is the hexadecimal value.

Thus if you issue SELECT c1 FROM t1 WHERE c1 LIKE 'ab%', you receive the following output:

```
ab
abe1
abe1s
```

If you issue SELECT c1 FROM t1 WHERE c1 LIKE 'A%', you receive the following output:

```
Abe1
Ab
ABEL
```



If you issue `SELECT c1 FROM t1 ORDER BY c1`, you receive the following:

```
ab
Ab
abel
Abe1
ABEL
abels
```

Thus, you may want to consider using the scalar function `TRANSLATE()`, as well as the `CREATE DATABASE` API. Note that you can only specify a collating sequence using the `CREATE DATABASE` API. You cannot specify a collating sequence from the Command Line Processor. For information on the `TRANSLATE()` function, see the *SQL Reference*. For information on the `CREATE DATABASE` API see the *Administrative API Reference*.

You can also use the `UCASE` function as follows, but note that DB2 performs a table scan instead of using an index for the select:

```
SELECT * FROM EMP WHERE UCASE(JOB) = 'NURSE'
```

### Collating Sequence Sort Order: EBCDIC and ASCII Example

The order in which data in a database is sorted depends on the collating sequence defined for the database. For example, suppose that database A uses the EBCDIC code page's default collating sequence and that database B uses the ASCII code page's default collating sequence. Sort orders at these two databases would differ, as shown in Figure 7.

```
SELECT.....
```

```
ORDER BY COL2
```

EBCDIC-Based Sort

ASCII-Based Sort

COL2

COL2

----

----

V1G

7AB

Y2W

V1G

7AB

Y2W

*Figure 7. Example of How a Sort Order in an EBCDIC-Based Sequence Differs from a Sort Order in an ASCII-Based Sequence*

Similarly, character comparisons in a database depend on the collating sequence defined for that database. So if database A uses the EBCDIC code page's default collating sequence and database B uses the ASCII code page's default collating sequence, the results of character comparisons at the two databases would differ. Figure 8 on page 398 illustrates the difference.

```

SELECT.....
 WHERE COL2 > 'TT3'

EBCDIC-Based Results ASCII-Based Results

COL2 COL2
---- ----
TW4 TW4
X72 X72
39G

```

*Figure 8. Example of How a Comparison of Characters in an EBCDIC-Based Sequence Differs from a Comparison of Characters in an ASCII-Based Sequence*

If you are creating a federated database, consider specifying that your collating sequence matches the collating sequence at a data source. This approach will maximize “push-down” opportunities and possibly increase query performance. See the *Administration Guide, Performance* for more information on the relationship between pushdown analysis, collating sequences, and query performance.

### Specifying a Collating Sequence

The collating sequence for a database is specified at database creation time. Once the database has been created, the collating sequence cannot be changed.

The CREATE DATABASE API accepts a data structure called the Database Descriptor Block (SQLEDBDESC). You can define your own collating sequence within this structure.

To specify a collating sequence for a database:

- Pass the desired SQLEDBDESC structure, or
- Pass a NULL pointer. The collating sequence of the operating system (based on current country code and code page) is used. This is the same as specifying SQLDBCSS equal to SQL\_CS\_SYSTEM (0).

The SQLEDBDESC structure contains:

#### SQLDBCSS

A 4-byte integer indicating the source of the database collating sequence. Valid values are:

#### SQL\_CS\_SYSTEM

The collating sequence of the operating system (based on current country code and code page) is used.

**SQL\_CS\_USER**

The collating sequence is specified by the value in the SQLDBUDC field.

**SQL\_CS\_NONE**

The collating sequence is the identity sequence. Strings are compared byte for byte, starting with the first byte, using a simple code point comparison.

**Note:** These constants are defined in the SQLENV include file.

**SQLDBUDC**

A 256-byte field. The nth byte contains the sort weight of the nth character in the code page of the database. If SQLDBCSS is not equal to SQL\_CS\_USER, this field is ignored.

**Sample Collating Sequences:** Several sample collating sequences are provided (as include files) to facilitate database creation using the EBCDIC collating sequences instead of the default workstation collating sequence.

The collating sequences in these include files can be specified in the SQLDBUDC field of the SQLEDBDESC structure. They can also be used as models for the construction of other collating sequences.

For information on the include files that contain collating sequences, see the following sections:

- For C/C++, “Include Files” on page 493
- For COBOL, “Include Files” on page 570
- For FORTRAN, “Include Files” on page 592.

**General Concerns:** Once a collating sequence is defined, all future character comparisons for that database will be performed with that collating sequence. Except for character data defined as FOR BIT DATA or BLOB data, the collating sequence will be used for all SQL comparisons and ORDER BY clauses, and also in setting up indexes and statistics. For more information on how the database collating sequence is used, see the section on *String Comparisons* in the *SQL Reference*.

Potential problems may occur in the following cases:

- An application merges sorted data from a database with application data that was sorted using a different collating sequence.
- An application merges sorted data from one database with sorted data from another, but the databases have different collating sequences.

- An application makes assumptions about sorted data that are not true for the relevant collating sequence. For example, numbers collating lower than alphabets might or might not be true for a particular collating sequence.

A final point to remember is that the results of any sort based on a direct comparison of code points of characters will only match the results of a query ordered using an identity collating sequence.

**Federated Database Concerns:** Your choice of database collating sequence can affect federated system performance. If a data source uses the same collating sequence as the DB2 federated database, DB2 can push-down order-dependent processing involving character data to the data source. If a data source collating sequence does not match DB2's, data is retrieved and all order-dependent processing on character data is done locally (which can slow performance).

To determine if a data source and DB2 have the same collating sequence, consider the following factors:

- National language support  
The collating sequence is related to the language supported on a server. Compare DB2 NLS information to data source NLS information.
- Data source characteristics  
Some data sources are created using case-insensitive collating sequences, which can yield different results from DB2 in order-dependent operations.
- Customization  
Some data sources provide multiple options for collating sequences or allow the collating sequence to be customized.

Choose the collating sequence for a DB2 federated database based on the mix of data sources that will be accessed from that database. For example:

- If a DB2 database will access mostly Oracle databases with the same code page (NLS) as DB2, specify SQL\_CS\_NONE for the SQLDBCSS structure (Oracle databases use an equivalent collating sequence).
- If a DB2 database will access only DB2 UDB databases, ensure that you match SQLDBCSS values.

For information on how to set up a MVS collating sequence, you should refer to the *Administrative API Reference* for samples under the “sqlecrea” topic. These samples contain collation tables for EBCDIC 500, 37, and 5026/5035 code pages.

After you set the collating sequence for the DB2 database, ensure that you set the **collating\_sequence** server option for each data source server. The

collating\_sequence option indicates if the collating sequence of a given data source server matches the collating sequence of the DB2 database.

Set the collating\_sequence option to “Y” if the collating sequences match. This setting allows the DB2 optimizer to consider order-dependent processing at a data source, which can improve performance. However, if the data source collating sequence is not the same as the DB2 database collating sequence, you can receive incorrect results. For example, if your plan uses merge joins, the DB2 optimizer will push-down ordering operations to the data sources as much as possible. If the data source collating sequence is not the same, the join results may not have a correct result set.

Set the collating\_sequence option to “N” if the collating sequences do not match. Use this value when data source collating sequences differ from DB2 or when the data source collating operations might be case insensitive. For example, in a case insensitive data source with an English code page, TOLLESON, ToLLeSoN, and tolleson would all be considered equal. Set the collating\_sequence option to “N” if you are not sure that the collating sequence at the data source is identical to the DB2 collating sequence.

## Deriving Code Page Values

The **application code page** is derived from the active environment when the database connection is made. If the DB2CODEPAGE registry variable is set, its value is taken as the application code page. However, it is not necessary to set the DB2CODEPAGE registry variable because DB2 will determine the appropriate code page value from the operating system. Setting the DB2CODEPAGE registry variable to incorrect values may cause unpredictable results.

The **database code page** is derived from the value specified (explicitly or by default) at the time the database is created. The following defines how the *active environment* is determined in different operating environments, for example:

**UNIX** In UNIX-based environments, the active environment is determined from the locale setting, which includes information about language, territory and code set.

**OS/2** In OS/2, primary and secondary code pages are specified in the CONFIG.SYS file. You can use the chcp command to display and dynamically change code pages within a given session.

### Windows 32-bit operating systems

For all Windows 32-bit operating systems, if the DB2CODEPAGE environment variable is not set, the code page is derived from the ANSI code page setting in the Registry.

For a complete list of environment mappings for code page values, see Table 44 on page 695.

## Deriving Locales in Application Programs

Locales are implemented one way in Windows and another in UNIX-based systems. In UNIX-based systems there are two locales:

- The environment locale allows you to specify the language, currency symbol, and so on, that you want to use.
- The program locale contains the current language, currency symbol, and so on, of a program that is executing.

In Windows, the cultural preferences can be set through Regional Settings of the Control Panel. However, there is no environmental locale like the one on UNIX systems.

When your program is started, it gets a default C locale. It does **not** get a copy of the environment locale. If you set the program locale to any locale other than “C”, DB2 Universal Database uses your current program locale to determine the code page and territory settings for your application environment. Otherwise, these values are obtained from the operating system environment. You should note that `setlocale()` is not thread-safe, and if you issue `setlocale()` from within your application, the new locale is set for the entire process.

### How DB2 Derives Locales

With UNIX, the active locale used by DB2 is determined from the LC\_CTYPE portion of the locale. For details, see the NLS documentation for your operating system.

- If LC\_CTYPE of the program locale has a value other than that of 'C', DB2 will use this value to determine the application code page by mapping it to its corresponding code page.
- If LC\_CTYPE has the value of 'C' (the 'C' locale), DB2 will set the program locale according to the environment locale using the `setlocale()` function.
- If LC\_CTYPE still has a value of 'C', DB2 will assume the default of the US English environment, and code page 819 (ISO 8859-1).
- If LC\_CTYPE's value is no longer 'C', its new value will be used to map to a corresponding code page. For information on the default locale for a particular platform, see Table 44 on page 695. Additional information may be found in the *Application Building Guide*.

## Programming Considerations

Constant character strings in static SQL statements are converted at bind time, from the application code page to the database code page, and will be used at execution time in this database code page representation. To avoid such conversions if they are not desired, you can use host variables in place of string constants.

If your program contains constant character strings, it is strongly recommended that application be precompiled, bound, compiled, and executed using the same code page. For a Unicode database, you should use host variables instead of using string constants. This is because data conversions by the server can occur in both the bind and the execution phases. This could be a concern if constant character strings are used within the program. These embedded strings are converted at bind time based on the code page which is in effect during the bind phase. Seven-bit ASCII characters are common to all the code pages supported by DB2 Universal Database and will not cause a problem. For non-ASCII characters, users should ensure that the same conversion tables are used by binding and executing with the same active code page. For a discussion of how applications determine the active code page, see “Deriving Code Page Values” on page 401.

Any external data obtained by the application will be assumed to be in the application code page. This includes data obtained from a file or from user input. Make sure that data from sources outside the application uses the same code page as the application.

If you use host variables that use graphic data in your C or C++ applications, there are special precompiler, application performance, and application design issues you need to consider. For a detailed discussion of these considerations, see “Handling Graphic Host Variables” on page 519. If you deal with EUC code sets in your applications, refer to “Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations” on page 419 for guidelines that you should consider.

### Coding SQL Statements

The coding of SQL statements is not language dependent. The SQL keywords must be typed as shown in this book, although they may be typed in uppercase, lowercase, or mixed case. The names of database objects, host variables and program labels that occur in an SQL statement cannot contain characters outside the extended character set supported by your code page. See the *SQL Reference* for more information about extended character sets.

The server does not convert file names. To code a file name, either use the ASCII invariant set, or provide the path in the hexadecimal values that are physically stored in the file system.

In a multi-byte environment, there are four characters which are considered special that do not belong to the invariant character set. These characters are:

- The double-byte percentage and double-byte underscore characters used in LIKE processing. For further details concerning LIKE, refer to the *SQL Reference*.
- The double-byte space character, used for, among other things, blank padding in graphic strings.
- The double-byte substitution character, used as a replacement during character conversion when no mapping exists between a source code page and a target code page.

The code points for each of these characters, by code page is as follows:

Table 13. Code Points for Special Double-byte Characters

| Code Page | Double-Byte Percentage | Double-Byte Underscore | Double-byte Space | Double-Byte Substitution Character |
|-----------|------------------------|------------------------|-------------------|------------------------------------|
| 932       | X'8193'                | X'8151'                | X'8140'           | X'FCFC'                            |
| 938       | X'8193'                | X'8151'                | X'8140'           | X'FCFC'                            |
| 942       | X'8193'                | X'8151'                | X'8140'           | X'FCFC'                            |
| 943       | X'8193'                | X'8151'                | X'8140'           | X'FCFC'                            |
| 948       | X'8193'                | X'8151'                | X'8140'           | X'FCFC'                            |
| 949       | X'A3A5'                | X'A3DF'                | X'A1A1'           | X'AFFE'                            |
| 950       | X'A248'                | X'A1C4'                | X'A140'           | X'C8FE'                            |
| 954       | X'A1F3'                | X'A1B2'                | X'A1A1'           | X'F4FE'                            |
| 964       | X'A2E8'                | X'A2A5'                | X'A1A1'           | X'FDFF'                            |
| 970       | X'A3A5'                | X'A3DF'                | X'A1A1'           | X'AFFE'                            |
| 1381      | X'A3A5'                | X'A3DF'                | X'A1A1'           | X'FEFE'                            |
| 1383      | X'A3A5'                | X'A3DF'                | X'A1A1'           | X'A1A1'                            |
| 13488     | X'FF05'                | X'FF3F'                | X'3000'           | X'FFFD'                            |

**UCS-2 Considerations:** For a UCS-2 database, the GRAPHIC space is X'0020' which is different from the “Double-byte Space” of X'3000' used for CCSID 13488. This difference should be taken into consideration when comparing data from a EUC database to data from a UCS-2 database. Note that in a UCS-2 database, Unicode representations of ASCII percent and an ASCII underscore are used for pattern matching. DBCS percent and DBCS



underscore have no special meaning for a UCS-2 database. The DBCS substitution character is used to replace any EUC non-SBCS character as required. There is no concept of a three or four byte substitution character.

### **Coding Remote Stored Procedures and UDFs**

When coding stored procedures that will be running remotely, the following considerations apply:

- Data in a stored procedure must be in the database code page.
- Data passed to or from a stored procedure using an SQLDA with a character data type must really contain character data. Numeric data and data structures must never be passed with a character type if the client application code page is different from the database code page. This is because the server will convert all character data in an SQLDA. To avoid character conversion, you can pass data by defining it in binary string format by using a data type of BLOB or by defining the character data as FOR BIT DATA.

By default, when you invoke DB2 DARI stored procedures and UDFs, they run under a default national language environment which may not match the database's national language environment. Consequently, using country or code page specific operations, such as the C wchar\_t graphic host variables and functions, may not work as you expect. You need to ensure that, if applicable, the correct environment is initialized when you invoke the stored procedure or UDF.

### **Package Name Considerations in Mixed Code Page Environments**

Package names are determined when you invoke the PRECOMPILE PROGRAM command or API. By default, they are generated based on the first 8-bytes of the application program source file (without the file extension) and are folded to upper case. Optionally, a name can be explicitly defined. Regardless of the origin of a package name, if you are running in an unequal code page environment, the characters for your package names should be in the invariant character set. Otherwise you may experience problems related to the modification of your package name. The database manager will not be able to find the package for the application or a client-side tool will not display the right name for your package.

A package name modification due to character conversion will occur if any of the characters in the package name, are not directly mapped to a valid character in the database code page. In such cases, a substitution character replaces the character that is not converted. After such a modification, the package name, when converted back to the application code page, may not match the original package name. An example of a case where this behavior is

undesirable is when you use the DB2 Database Director to list and work with packages. Package names displayed may not match the expected names.

To avoid conversion problems with package names, ensure that only characters are used which are valid under both the application and database code pages.

### Precompiling and Binding

At precompile/bind time, the precompiler is the executing application. The active code page when the database connection was made prior to the precompile request is used for precompiled statements, and any character data returned in the SQLCA.

### Executing an Application

At execution time, the active code page of the user application when a database connection is made is in effect for the duration of the connection. All data is interpreted based on this code page; this includes dynamic SQL statements, user input data, user output data, and character fields in the SQLCA.

### A Note of Caution

Failure to follow these guidelines may produce unpredictable results. These conditions cannot be detected by the database manager, so no error or warning message will result. For example, a C application contains the following SQL statements operating against a table T1 with one column defined as C1 CHAR(20):

- (0) EXEC SQL CONNECT TO GLOBALDB;
- (1) EXEC SQL INSERT INTO T1 VALUES ('a-constant');  
strcpy(sqlstmt, "SELECT C1 FROM T1 WHERE C1='a-constant');"
- (2) EXEC SQL PREPARE S1 FROM :sqlstmt;

Where:

application code page at bind time = **x**  
application code page at execution time = **y**  
database code page = **z**

At bind time, 'a-constant' in statement (1) is converted from code page **x** to code page **z**. This conversion can be noted as (x→z).

At execution time, 'a-constant' (x→z) is inserted into the table when statement (1) is executed. However, the WHERE clause of statement (2) will be executed with 'a-constant' (y→z). If the code points in the constant are such that the two conversions (x→z and y→z) yield different results, the SELECT in statement (2) will fail to retrieve the data inserted by statement (1).

## Conversion Between Different Code Pages

Ideally, for optimal performance, your applications should always use the same code page as your database. However, this is not always practical or possible. The DB2 products provide support for character conversion that allows your application and database to use different code pages. Characters from one code page must be mapped to the other code page in order to maintain meaning of the data.

**When Does Character Conversion Occur?:** Character conversion can occur in the following situations:

- When a client or application accessing a database is running in a code page that is different from the code page of the database.

This database conversion will occur on the database server machine for both conversions from the application code page to the database code page and from the database code page to the application code page.

You can minimize or eliminate client/server character conversion in some situations. For example, you could:

- Create a Windows NT database using code page 850 to match an OS/2 and Windows client application environment that predominately uses code page 850,

If a Windows ODBC application is used with the IBM DB2 ODBC driver in Windows database client, this problem may be alleviated by the use of the TRANSLATEDLL and TRANSLATEOPTION keywords in the `odbc.ini` or `db2cli.ini` file.

- Create a DB2 for AIX database using code page 850 to match an OS/2 and DOS client application environment that predominately uses code page 850.

**Note:** The DB2 for OS/2 Version 1.0 or Version 1.2 database server does not support character conversion between different code pages. Ensure that the code pages on server and client are compatible. See Table 44 for the code page conversions that are supported.

- When a client or application importing a PC/IXF file runs in a code page that is different from the file being imported.

This data conversion will occur on the database client machine before the client accesses the database server. Additional data conversion may take place if the application is running in a code page that is different from the code page of the database (as stated in the previous point).

Data conversion, if any, also depends on how the import utility was called. See the *Administration Guide* for more information.

- When DB2 Connect is used to access data on a host or AS/400 server. In this case the data receiver converts the character data. For example, data that is sent to DB2 for MVS/ESA is converted to the appropriate MVS

coded character set identifier (CCSID) by DB2 for MVS/ESA. The data sent back to the DB2 Connect machine from DB2 for MVS/ESA is converted by DB2 Connect. For more information, see the *DB2 Connect User's Guide*.

Character conversion will **not** occur for:

- File names. You should either use the ASCII invariant set for file names or provide the file name in the hexadecimal values that are physically stored in the file system. Note that if you include a file name as part of a SQL statement, it gets converted as part of the statement conversion.
- Data that is targeted for or comes from a column assigned the FOR BIT DATA attribute, or data used in an SQL operation whose result is FOR BIT or BLOB data. In these cases, the data is treated as a byte stream and no conversion occurs.<sup>1</sup> See the *SQL Reference* for unequal code page rules for assigning, comparing, and combining strings.
- A DB2 product or platform that does not support, or that does not have support installed, for the desired combination of code pages. In this case, an SQLCODE -332 (SQLSTATE 57017) is returned when you try to run your application.

**Character Substitutions During Conversions:** When your application converts from one code page to another, it is possible that one or more characters are not represented in the target code page. If this occurs, DB2 inserts a *substitution* character into the target string in place of the character that has no representation. The replacement character is then considered a valid part of the string. In situations where a substitution occurs, the SQLWARN10 indicator in the SQLCA is set to 'W'.

**Note:** Any character conversions resulting from using the WCHARTYPE CONVERT precompiler option will not flag a warning if any substitutions take place.

**Supported Character Conversions:** When data conversion occurs, conversion will take place from a **source code page** to a **target code page**.

The source code page is determined from the source of the data; data from the application has a source code page equal to the application code page, and data from the database has a source code page equal to the database code page.

The determination of target code page is more involved; where the data is to be placed, including rules for intermediate operations, is considered:

---

1. However, a literal inserted into a column defined as FOR BIT DATA could be converted if that literal was part of an SQL statement which was converted.

- If the data is moved directly from an application into a database, with no intervening operations, the target code page is the database code page.
- If the data is being imported into a database from a PC/IXF file, there are two character conversion steps:
  1. From the PC/IXF file code page (source code page) to the application code page (target code page)
  2. From the application code page (source code page) to the database code page (target code page).

Exercise caution in situations where two conversion steps might occur. Make sure you follow the supported character conversions listed in Table 44 to avoid a possible loss of character data. Additionally, within each group, only characters which exist in both the source and target code page have meaningful conversions. Other characters are used as “substitutions” and are only useful for converting from the target code page back to the source code page (and may not necessarily provide meaningless conversions in the two-step conversion process mentioned above). Such problems are avoided if the application code page is the same as the database code page.

- If the data is derived from operations performed on character data, where the source may be any of the application code page, the database code page, FOR BIT DATA, or for BLOB data, data conversion is based on a set of rules. Some or all of the data items may have to be converted to an intermediate result, before the final target code page can be determined. See the *SQL Reference* for a summary of these rules and for specific application with individual operators and predicates.

Table 44 on page 695 shows the code pages that are supported by DB2 Universal Database. The values under the heading “Group” can be used to determine where conversions are supported. Any code page can be converted to any other code page that is listed in the same IBM-defined language group. For example, code page 437 can be converted to 37, 819, 850, 1051, 1252, or 1275.

**Note:** Character string conversions between multi-byte code pages, for example DBCS and EUC, may result in either an increase or a decrease in the length of the string.

**Character Conversion Expansion Factor:** When your application successfully completes an attempt to connect to a DB2 database server, you should consider the following fields in the returned SQLCA:

- The second token in the SQLERRMC field (tokens are separated by X'FF') indicates the code page of the database. The ninth token in the SQLERRMC field indicates the code page of the application. Querying the application's

code page and comparing it to the database's code page informs the application whether it has established a connection which will undergo character conversions.

- The first and second entries in the SQLERRD array. SQLERRD(1) contains an integer value equal to the maximum expected expansion or contraction factor for the length of mixed character data (CHAR data types) when converted to the database code page from the application code page. SQLERRD(2) contains an integer value equal to the maximum expected expansion or contraction factor for the length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. Refer to the *SQL Reference* for details on using the CONNECT statement.

The considerations for graphic string data should not be a factor in unequal code page situations. Each string always has the same number of characters, regardless of whether the data is in the application or the database code page.

See “Unequal Code Page Situations” on page 425 for information on dealing with unequal code page situations.

## Unicode/UCS-2 and UTF-8 Support in DB2 UDB

These two standards are documented here.

### Introduction

The Unicode character encoding standard is a fixed-length, character encoding scheme that includes characters from almost all the living languages of the world. Unicode characters are usually shown as “U+xxxx” where xxxx is the hexadecimal code of the character.

Each character is 16 bits (2 bytes) wide, regardless of the language. While the resulting 65 000 code elements are sufficient for encoding most of the characters of the major languages of the world, the Unicode standard also provides an extension mechanism, that allows for encoding as many as a million more characters. This extension reserves a range of code values (U+D800 to U+D8FF, known as “surrogates”) for encoding some 32-bit characters as two successive code elements.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) 10646 standard (ISO/IEC 10646) specifies the Universal Multiple-Octet Coded Character Set (UCS) that has a 2-byte version (UCS-2) and a 4-byte version (UCS-4). The 2-byte version of this ISO standard UCS-2 is identical to Unicode without surrogates. ISO 10646 also

defines an extension technique, for encoding some UCS-4 codes in a UCS-2 encoded string. This extension called UTF-16, is identical to Unicode with surrogates.

DB2 UDB supports UCS-2, that is, Unicode without surrogates.

**UTF-8:** With UCS-2 or Unicode encoding, ASCII and control characters are also two bytes long, and the lead byte is zero. For example, NULL is U+0000 and CAPITAL LETTER A is represented by U+0041. This could be a major problem for ASCII-based applications and ASCII file systems, because in a UCS-2 strings, extraneous NULLs may appear anywhere in the string. A transformation algorithm, known as UTF-8, can be used to circumvent this problem for programs that rely on ASCII code being invariant.

UTF-8 (UCS Transformation Format 8), is an algorithmic transformation which transforms fixed-length UCS-4 characters into variable-length byte strings. In UTF-8, ASCII characters are represented by their usual single-byte codes, but non-ASCII characters in UCS-2 become two or three bytes long. In other words, UTF-8 transforms UCS-2 characters to a multi-byte codeset, for which ASCII is invariant. The number of bytes for each UCS-2 character in UTF-8 format can be determined from the following table:

| UCS-2 (hex)  | UTF-8 (binary)             | Description  |
|--------------|----------------------------|--------------|
| -----        | -----                      | -----        |
| 0000 to 007F | 0xxxxxxx                   | ASCII        |
| 0080 to 07FF | 110xxxxx 10xxxxxx          | up to U+07FF |
| 0800 to FFFF | 1110xxxx 10xxxxxx 10xxxxxx | other UCS-2  |

NOTE: The range D800 to DFFF is to be excluded from treatment by the third row of this table which governs the UCS-4 range 0000 0800 to 0000 FFFF.

In all the above, a series of x's indicate the UCS bit representation of the character. For example, U0080 transforms into 11000010 10000000.

### UCS-2/UTF-8 Implementation in DB2 UDB

**Code Page/CCSID Numbers:** Within IBM, the UCS-2 code page has been registered as code page 1200. All code pages are defined with growing character sets, that is, when new characters are added to a code page, the code page number does not change. Code page 1200 always refers to the current version of Unicode/UCS-2, and has been used for UCS-2 support in DB2 UDB.

A specific repertoire of the UCS standard, as defined by Unicode 2.0 and ISO/IEC 10646-1, has also been registered within IBM as CCSID 13488. This CCSID (13488) has been used internally by DB2 UDB for storing graphic string data in euc-Japan and euc-Taiwan databases. CCSID 13488 and code



page 1200 both refer to UCS-2, and are handled the same way except for the value of their “double-byte” (DBCS) space:

| CP/CCSID | Single Byte (SBCS) space | Double Byte (DBCS) space |
|----------|--------------------------|--------------------------|
| -----    | -----                    | -----                    |
| 1200     | N/A                      | U+0020                   |
| 13488    | N/A                      | U+3000                   |

NOTE: In a UCS-2 database, U+3000 has no special meaning.

Regarding the conversion tables, since code page 1200 is a superset of CCSID 13488, the exact same (superset) tables are used for both.

In IBM, UTF-8 has been registered as CCSID 1208 with growing character set (sometimes also referred to as code page 1208). As new characters are added to the standard, this number (1208) will not change either. 1208 is used as the multi-byte code page number for DB2’s UCS-2/UTF-8 support.

DB2 UDB supports UCS-2 as a new multi-byte code page. The MBCS code page number is 1208, which is the database code page number, and the code page of character string data within the database. The double-byte code page number (for UCS-2) is 1200 which is the code page of graphic string data within the database. When a database is created in UCS-2/UTF-8, CHAR, VARCHAR, LONG VARCHAR, and CLOB data, are stored in UTF-8, and GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB data, are stored in UCS-2. We will simply refer to this as a UCS-2 database.

**Creating a UCS-2 Database:** By default, databases are created in the code page of the application creating them. Therefore, if you create your database from a UTF-8 client (for example, the UNIVERSAL locale of AIX), or if DB2CODEPAGE environment variable on the client is set to 1208, your database will be created as a UCS-2 database. Alternatively, you can explicitly specify “UTF-8” as the CODESET name, and use any valid two letter TERRITORY code supported by DB2 UDB. Refer to the *Administration Guide*, *Design and Implementation*, and the appendix on “NLS Support” for a complete list of territory codes supported by DB2 UDB.

For example, to create a UCS-2 database from the CLP, with the territory code for United States, issue:

```
DB2 CREATE DATABASE dbname USING CODESET UTF-8 TERRITORY US
```

To create a UCS-2 database using the *sqlecrea* API, you should set the values in *sqledbcountryinfo* accordingly. For example, set *sqledbcountryinfo* to “UTF-8”, and *sqldblocale* to any valid territory code (for example, “US”).



The default collation sequence for a UCS-2 database is IDENTITY, which provides UCS-2 code point order. Therefore, by default, all UCS-2/UTF-8 characters are ordered and compared according to their UCS-2 code point sequence.

All cultural-sensitive parameters such as date/time format, decimal separator, and others, are based on the current territory of the client.

A UCS-2 database allows connection from every single-byte and multi-byte code page supported by DB2 UDB. Code page character conversions between client's code page and UTF-8 are automatically performed by the database manager. Data in graphic string types, is always in UCS-2 and does not go through code page conversions. The Command Line Processor (CLP) environment is an exception. If you SELECT graphic string (UCS-2) data from the CLP, the returned graphic string data is converted (by the CLP) from UCS-2 to the code page of your client environment.

Every client is limited by the character repertoire, the input methods, and the fonts supported by its environment, but the UCS-2 database itself accepts and stores all UCS-2 characters. Therefore, every client usually works with a subset of UCS-2 characters, but the database manager allows the entire repertoire of UCS-2 characters.

When characters are converted from a local code page to UTF-8, there is a possibility of expansion in the number of bytes. There is no expansion for ASCII characters, but other UCS-2 characters expand by a factor of two or three. The number of bytes of each UCS-2 character in UTF-8 format can be determined from the above table (section about UTF-8).

**Data Types:** All data types supported by DB2 UDB, are also supported in a UCS-2 database. In particular, graphic string data, is supported for UCS-2 database and is stored in UCS-2/Unicode. Every client, including SBCS clients, can work with graphic string data types in UCS-2/Unicode when connected to a UCS-2 database.

A UCS-2 database is like any MBCS database where character string data is measured in number of bytes. When working with character string data in UTF-8, one should not assume that each character is one byte. In multi-byte UTF-8 encoding, each ASCII character is one byte, but non-ASCII characters take two or three bytes each. This should be taken into account when defining CHAR fields. Depending on the ratio of ASCII to non-ASCII characters, a CHAR field of size  $n$  bytes, can contain anywhere from  $n/3$  to  $n$  characters.

Using character string UTF-8 encoding versus graphic string UCS-2 data type also has an impact on the total storage requirements. For a situation where the majority of characters are ASCII, with some non-ASCII characters in between,

storing UTF-8 data may be a better alternative because the storage requirements are closer to one byte per character. On the other hand, for situations where the majority of characters are non-ASCII characters that expand to three-byte UTF-8 sequences (for example ideographic characters), the UCS-2 graphic-string format may be a better alternative because every UCS-2 character requires exactly two bytes, rather than three bytes for each corresponding character in UTF-8 format.

SQL scalar functions that operate on character strings, such as LENGTH, SUBSTR, POSSTR, MAX, MIN, and the like, in MBCS environments operate on number of “bytes” rather than number of “characters”. The behavior is the same in a UCS-2 database but you should take extra care when specifying offsets and lengths for a UCS-2 database because these values are always defined in the context of the database code page. That is, in the case of UCS-2 database should be defined in UTF-8. Since some single-byte characters require more than one byte in UTF-8, SUBSTR indexes that are valid for a single-byte database may not be valid for a UCS-2 database. If you specify incorrect indexes, you will get SQLCODE -191, SQLSTATE 22504. Refer to the *SQL Reference* for a description of the behavior of these functions.

SQL CHAR data types are supported by C language’s char data type in user programs. SQL GRAPHIC data types are supported by sqldbcchar in user C programs. Note that, for a UCS-2 database, sqldbcchar data is always in big-endian (high byte first) format. When an application program is connected to a UCS-2 database, character string data is converted between the application code page and UTF-8 by DB2 UDB, but graphic string data is always in UCS-2. However, you can set the WCHARTYPE pre-compiler option to CONVERT, if you want to convert UCS-2 data to your program’s wchar\_t. See “Handling Graphic Host Variables” on page 519 for more information.

**Identifiers:** In a UCS-2 database, all identifiers are in multi-byte UTF-8. Therefore, it is possible to use any UCS-2 character in identifiers where the use of a character in the extended character set (for example, an accented character, or a multi-byte character) is allowed by DB2 UDB. Please refer to the appendix “Naming Rules” in the *Administration Guide* for details of which identifiers allow use of extended characters.

Clients can enter any character which is supported by their SBCS/MBCS environment, and all the characters in the identifiers will be converted to UTF-8 by the database manager. Two points need to be taken into account when specifying National Language characters in identifiers in a UCS-2 database:

1. Each non-ASCII character takes two or three bytes. Therefore, an n-byte identifier, can only hold somewhere between  $n/3$  and  $n$  characters, depending the ratio of ASCII to non-ASCII characters. If you have only

one or two non-ASCII (for example, accented) characters, the limit is closer to  $n$  characters, while for an identifier which is completely non-ASCII (for example, in Japanese), only  $n/3$  characters can be used.

2. If identifiers are to be entered from different client environments, they should be defined using the common subset of characters available to those clients. For example, if a UCS-2 database is to be accessed from Latin-1, Arabic, and Japanese environments, all identifiers should realistically be limited to ASCII.

**UCS-2 Literals:** UCS-2 literals can be specified in two ways:

1. As a GRAPHIC string constant using the G'...' or N'...' format as described in the *SQL Reference*, Chapter 3 "Language Elements", the section "Constants", the subsection "Graphic String Constants." Any literal specified in this way will be converted by the database manager from the application code page to UCS-2.
2. As a UCS-2 hexadecimal string, using the UX'...' or GX'...' format. The constant specified between quotes after UX or GX must be a multiple of 4 hexadecimal digits. Each four digits represent one UCS-2 code point.

When using the Command Line Processor (CLP), the first method is easier if the UCS-2 character exists in the local application code page (for example, for entering any code page 850 character from a terminal that is using code page 850). The second method should be used for characters which are outside the application code page repertoire (for example, for specifying Japanese characters from a terminal that is using code page 850).

**Pattern Matching in a UCS-2 Database:** Pattern matching is one area where the behavior of existing MBCS databases is slightly different from the behavior of a UCS-2 database.

For MBCS databases in DB2 UDB, the current behavior is as follows: If the match-expression contains MBCS data, the pattern can include both SBCS and MBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one MBCS character.
- A percent (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

If the match-expression contains graphic string DBCS data, the expressions contain only DBCS characters. The special characters in the pattern are interpreted as follows:

- A DBCS underscore refers to one DBCS character.
- A DBCS percent refers to a string of zero or more DBCS characters.

In a UCS-2 database, there is really no distinction between “single-byte” and “double-byte” characters; every UCS-2 character occupies two bytes. Although the UTF-8 format is a “mixed-byte” encoding of UCS-2 characters, there is no real distinction between SBCS and MBCS characters in UTF-8. Every character is a UCS-2 character, irrespective of its number of bytes in UTF-8 format. When specifying a character string, or a graphic string expression, an underscore refers to one UCS-2 characters and a percent refers to a string of zero or more UCS-2 characters.

On the client side, the character string expressions are in the code page of the client, and will be converted to UTF-8 by the database manager. SBCS client code pages, do not have any DBCS percent or DBCS underscore, but every supported code page contains a single-byte percent (corresponding to U+0025) and a single-byte underscore (corresponding to U+005F). The interpretation of special characters for a UCS-2 database is as follows:

- An SBCS underscore (corresponding to U+0025) refers to one UCS-2 character in a graphic string expression, or to one UTF-8 character in a character string expression.
- An SBCS percent (corresponding to U+005F) refers to a string of zero or more UCS-2 characters in a graphic string expression, or to a string of zero or more UTF-8 characters in a character string expression.

DBCS code pages, additionally support a DBCS percent sign (corresponding to U+FF05) and a DBCS underscore (corresponding to U+FF3F). These characters have no special meaning for a UCS-2 database.

For the optional “escape-expression” which specifies a character to be used to modify the special meaning of the underscore and percent characters, only ASCII characters, or characters that expand into a two-byte UTF-8 sequence, are supported. If you specify an ESCAPE character which expands to a three-byte UTF-8 value, you will get an error message (SQL0130N error, SQLSTATE 22019).

**IMPORT/EXPORT/LOAD Considerations:** The DEL, ASC, and PC/IXF file formats are supported for a UCS-2 database as described in this section. The WSF format is not supported.

When exporting from a UCS-2 database to an ASCII delimited (DEL) file, all character data is converted to the application code page. Both character string and GRAPHIC string data are converted to the same SBCS or MBCS code page of the client. This is the existing behavior for the export of any database, and cannot be changed because the entire ASCII delimited file can have only one code page. Therefore, if you export to an ASCII delimited file, only those UCS-2 characters that exist in your application code page would be saved. Other characters are replaced with the default substitution character for the

application code page. For UTF-8 clients (code page 1208) there is no data loss because all UCS-2 characters are supported by UTF-8 clients.

When importing from an ASCII file (DEL or ASC) to a UCS-2 database, character string data is converted from the application code page to UTF-8, and GRAPHIC string data is converted from the application code page to UCS-2. There is no data loss. If you want to import ASCII data that has been saved under a different code page, you should switch the data file code page before issuing the import command. One way to accomplish this, is to set DB2CODEPAGE to the code page of the ASCII data file.

The range of valid ASCII delimiters, for SBCS and MBCS clients is identical to what is currently supported by DB2 UDB for these clients. The range of valid delimiters for UTF-8 clients is 0x01 to 0x7F, with the usual restrictions. Refer to the “IMPORT/EXPORT/LOAD Utility File Formats” appendix in the *Command Reference* for a complete list of these restrictions.

When exporting from a UCS-2 database to a PC/IXF file, character string data is converted to the SBCS/MBCS code page of the client. GRAPHIC string data is not converted and is stored in UCS-2 (code page 1200). There is no data loss.

When importing from an PC/IXF file to a UCS-2 database, character string data is assumed to be in the SBCS/MBCS code page stored in the PC/IXF header and GRAPHIC string data is assumed to be in the DBCS code page stored in the PC/IXF header. Character string data is converted by the IMPORT utility from the code page specified in the PC/IXF header to the code page of the client, and then from the client code page to UTF-8 (by the INSERT statement). GRAPHIC string data is converted by the IMPORT utility from the DBCS code page specified in the PC/IXF header directly to UCS-2 (code page 1200).

LOAD directly places the data into the database and by default, assumes data in ASC or DEL files is in the code page of the database. Therefore, by default no code page conversion takes place for ASCII files. When the code page of the data file has been explicitly specified (using the MODIFIED BY codepage=x command parameter), LOAD uses this information to convert from the specified code page x into the database code page before loading the data. For PC/IXF files, LOAD always converts from the code pages specified in the IXF header to the database code page (1208 for CHAR, 1200 for GRAPHIC).

The code page of DBCLOB files (as specified using the MODIFIED BY lobsinfile command parameter) is always 1200 for UCS-2. The code page of the CLOB files is the same as the code page of the data files being imported, loaded or exported. For example, for load or import using PC/IXF format, the

CLOB file is assumed to be in the code page specified by the PC/IXF header. If the DBCLOB file is in ASC or DEL format, for LOAD the CLOB data is assumed to be in the code page of the database (unless explicitly specified otherwise using the MODIFIED BY codepage=x command parameter), and for IMPORT it is assumed to be in the client application code page.

The NOCHECKLENGTHS option is always set to TRUE for a UCS-2 database because in a UCS-2 database, any SBCS can be connected to the database for which there is no DBCS code pages; and also because character strings in UTF-8 format usually have different length than corresponding length in client code pages.

Refer to *Data Movement Utilities Guide and Reference* for more information on the LOAD, EXPORT, and IMPORT utilities.

**Incompatibilities:** An application connected to a UCS-2 database, the graphic string data is always in UCS-2 (code page 1200). For applications connected to non UCS-2 databases, the graphic string data is in the applications DBCS code page; or, not allowed if the application code page is SBCS. For example, when a 932 client is connected to a Japanese non UCS-2 database, then the graphic string data is in code page 301. But for the 932 client applications connected to a UCS-2 database, the graphic string data is in UCS-2. However, you can set the WCHARTYPE pre-compiler option to convert graphic string data to `wchar_t` of your applications in both cases. See “Handling Graphic Host Variables” on page 519 for more information.

## Running CLI/ODBC/JDBC/SQLJ Programs in a DBCS Environment

For details on running Java programs that access DB2 Universal Database in a double-byte character set (DBCS) environment, see DB2 Java - DBCS Support online (<http://www.software.ibm.com/data/db2/java/dbcsjava.html>). This web page currently contains the following information:

JDBC and SQLJ programs access DB2 using the DB2 CLI/ODBC driver and therefore use the same configuration file (`db2cli.ini`). The following entries must be added to this configuration file if you run Java programs that access DB2 Universal Database in a DBCS environment:

### **PATCH1 = 65536**

This forces the driver to manually insert a "G" in front of character literals which are in fact graphic literals. This PATCH1 value should always be set when working in a double byte environment.

### **PATCH1 = 64**

This forces the driver to NULL terminate graphic output strings. This is needed by Microsoft Access in a double byte environment. If you need to use this PATCH1 value as well then you would add the two

values together ( $64+65536 = 65600$ ) and set `PATCH1=65600`. See Note #2 below for more information about specifying multiple `PATCH1` values.

**PATCH2 = 7**

This forces the driver to map all graphic column data types to char column data type. This is needed in a double byte environment.

**PATCH2 = 10**

This setting should only be used in an EUC (Extended Unix Code) environment. It ensures that the CLI driver provides data for character variables (CHAR, VARCHAR, etc...) in the proper format for the JDBC driver. The data in these character types will not be usable in JDBC without this setting.

**Note:**

1. Each of these keywords is set in each database specific stanza of the `db2cli.ini` file. If you want to set them for multiple databases then you need to repeat them for each database stanza in `db2cli.ini`.
2. To set multiple `PATCH1` values you add the individual values and use the sum. To set `PATCH1` to both 64 and 65536 you would set `PATCH1=65600` ( $64+65536$ ). If you already have other `PATCH1` values set then replace the existing number with the sum of the existing number and the new `PATCH1` values you want to add.
3. To set multiple `PATCH2` values you specify them in a comma delimited string (unlike the `PATCH1` option). To set `PATCH2` values 1 and 7 you would set `PATCH2="1,7"`

For more information about setting these keywords refer to the *Installation and Configuration Supplement*.

---

## Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations

Extended UNIX Code (EUC) denotes a set of general encoding rules that can support from one to four character sets in UNIX-based operating environments. The encoding rules are based on the ISO 2022 definition for encoding 7-bit and 8-bit data in which control characters are used to separate some of the character sets. EUC is a means of specifying a collection of code sets rather than a code set encoding scheme. A code set based on EUC conforms to the EUC encoding rules but also identifies the specific character sets associated with the specific instances. For example, the IBM-eucJP code set for Japanese refers to the encoding of the Japanese Industrial Standard characters according to the EUC encoding rules. For a list of code pages which are supported, refer to your platform's *Quick Beginnings* book.



Database and client application support for graphic (pure double-byte character) data, while running under EUC code pages with character encoding that is greater than two bytes in length is limited. The DB2 Universal Database products implement strict rules for graphic data that require all characters to be exactly two bytes wide. These rules do not allow many characters from both the Japanese and Traditional Chinese EUC code pages. To overcome this situation, support is provided at both the application level and the database level to represent Japanese and Traditional Chinese EUC graphic data using another encoding scheme.

A database created under either Japanese or Traditional Chinese EUC code pages will actually store and manipulate graphic data using the ISO 10646 UCS-2 code set, a double-byte encoding scheme which is a proper subset of the full ISO 10646 standard. Similarly, an application running under those code pages will send graphic data to the database server as UCS-2 encoded data. With this support, applications running under EUC code pages can access the same types of data as those running under DBCS code pages. For additional information regarding EUC environments, refer to the *SQL Reference*. The IBM-defined code page identifier associated with UCS-2 is 1200, and the CCSID number for the same code page is 13488. Graphic data in an eucJP or eucTW database uses the CCSID number 13488. In a UCS-2 database, use the code page number 1200 for GRAPHIC data.

The ISO 10646 standard specifies the encoding of a number of *combining* characters that are necessary in several scripts, such as Indic, Thai, Arabic and Hebrew. These characters can also be used for a productive generation of characters in Latin, Cyrillic, and Greek scripts. However their presence creates a possibility of an alternative coding for the same text. Although the coding is unambiguous and data integrity is preserved, a processing of text that contains combining characters is more complex. To provide for conformance of applications that choose not to deal with the combining characters, ISO 10646 defines three implementation levels:

**Level 1.**

Does not allow combining characters.

**Level 2.**

Only a subset of combining characters are allowed.

**Level 3.**

All combining characters are allowed.

DB2 Universal Database supports the entire set of UCS-2 characters, including all the combining characters, but does not perform any composition or decomposition of characters. For more information on the Unicode standard,



refer to the *Unicode Standard Version 2.0* from Addison-Wesley. For more information about UCS-2, refer to *ISO/IEC 10646-1* from the International Standard Organization.

If you are working with applications or databases using these character sets you may need to consider dealing with UCS-2 encoded data. When converting UCS-2 graphic data to the application's EUC code page, there is the possibility of an increase in the length of data. For details of data expansion, see "Character Conversion Expansion Factor" on page 409. When large amounts of data are being displayed, it may be necessary to allocate buffers, convert, and display the data in a series of fragments.

The following sections discuss how to handle data in this environment. For these sections, the term EUC is used to refer only to Japanese and Traditional Chinese EUC character sets. Note that the discussions do not apply to DB2 Korean or Simplified-Chinese EUC support since graphic data in these character sets is represented using the EUC encoding.

### **Mixed EUC and Double-Byte Client and Database Considerations**

The administration of database objects in mixed EUC and double-byte code page environments is complicated by the possible expansion or contraction in the length of object names as a result of conversions between the client and database code page. In particular, many administrative commands and utilities have documented limits to the lengths of character strings which they may take as input or output parameters. These limits are typically enforced at the client, unless documented otherwise. For example, the limit for a table name is 128 bytes. It is possible that a character string which is 128 bytes under a double-byte code page is larger, say 135 bytes, under an EUC code page. This hypothetical 135-byte table name would be considered invalid by such commands as REORGANIZE TABLE if used as an input parameter despite being valid in the target double-byte database. Similarly, the maximum permitted length of output parameters may be exceeded, after conversion, from the database code page to the application code page. This may cause either a conversion error or output data truncation to occur.

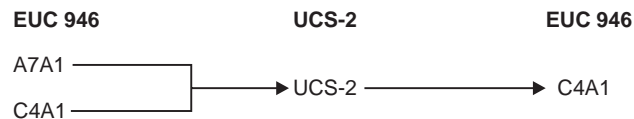
If you expect to use administrative commands and utilities extensively in a mixed EUC and double-byte environment, you should define database objects and their associated data with the possibility of length expansion past the supported limits. Administering an EUC database from a double-byte client imposes fewer restrictions than administering a double-byte database from an EUC client. Double-byte character strings will always be equal in length or shorter than the corresponding EUC character string. This will generally lead to less problems caused by enforcing the character string length limits.

**Note:** In the case of SQL statements, validation of input parameters is not conducted until the entire statement has been converted to the database code page. Thus you can use character strings which may be technically longer than allowed when they are represented in the client code page, but which meet length requirements when represented in the database code page.

## Considerations for Traditional Chinese Users

Due to the standards definition for Traditional Chinese, there is a side effect that you may encounter when you convert some characters between double-byte or EUC code pages and UCS-2. There are 189 characters (consisting of 187 radicals and 2 numbers) that share the same UCS-2 code point, when converted, as another character in the code set. When these characters are converted back to double-byte or EUC, they are converted to the code point of the same character's ideograph, with which it shares the same UCS-2 code point, rather than back to the original code point. When displayed, the character appears the same, but has a different code point. Depending on your application's design, you may have to take this behavior into account.

As an example, consider what happens to code point A7A1 in EUC code page 964, when it is converted to UCS-2 and then converted back to the original code page, EUC 946:



Thus, the original code points A7A1 and C4A1 end up as code point C4A1 after conversion.

If you require the code page conversion tables for EUC code pages 946 (Traditional Chinese EUC) or 950 (Traditional Chinese Big-5) and UCS-2, see the online Product and Service Technical Library (<http://www.software.ibm.com/data/db2/library/>).

## Developing Japanese or Traditional Chinese EUC Applications

When developing EUC applications, you need to consider the following items:

- Graphic Data Handling
- Developing for Mixed Code Set Environments

For additional considerations for stored procedures, see “Considerations for Stored Procedures” on page 424. Additional language-specific application development issues are discussed in:

- “Japanese or Traditional Chinese EUC, and UCS-2 Considerations” on page 524 (for C and C++).
- “Japanese or Traditional Chinese EUC, and UCS-2 Considerations” on page 590 (for COBOL).
- “Japanese or Traditional Chinese EUC, and UCS-2 Considerations” on page 605 (for FORTRAN).
- “Japanese or Traditional Chinese EUC Considerations” on page 625 (for REXX).

## Graphic Data Handling

This section discusses EUC application development considerations in order to handle graphic data. This includes handling graphic constants, and handling graphic data in UDFs, stored procedures, DBCLOB files, as well as collation.

**Graphic Constants:** Graphic constants, or literals, are actually classified as mixed character data as they are part of an SQL statement. Any graphic constants in an SQL statement from a Japanese or Traditional Chinese EUC client are implicitly converted to the graphic encoding by the database server. You can use graphic literals that are composed of EUC encoded characters in your SQL applications. An EUC database server will convert these literals to the graphic database code set which will be UCS-2. Graphic constants from EUC clients should never contain single-width characters such as CS0 7-bit ASCII characters or Japanese EUC CS2 (Katakana) characters.

For additional information on graphic constants, refer to the *SQL Reference*.

**Considerations for UDFs:** UDFs are invoked at the database server and are meant to deal with data encoded in the same code set as the database. In the case of databases running under the Japanese or Traditional Chinese code set, mixed character data is encoded using the EUC code set under which the database is created. Graphic data is encoded using UCS-2. This means that UDFs need to recognize and handle graphic data which will be encoded with UCS-2.

For example, you create a UDF called VARCHAR which converts a graphic string to a mixed character string. The VARCHAR function has to convert a graphic string encoded as UCS-2 to an EUC representation if the database is created under the EUC code sets.

**Considerations for Stored Procedures:** A stored procedure, running under either a Japanese or Traditional Chinese EUC code set, must be prepared to recognize and handle graphic data encoded using UCS-2. When running these code sets, graphic data received or returned through the stored procedure's input/output SQLDA is encoded using UCS-2.

### **Considerations for DBCLOB Files**

There are two important considerations for DBCLOB files:

- The DBCLOB file data is assumed to be in the EUC code page of the application. For EUC DBCLOB files, data is converted to UCS-2 at the client on read, and from UCS-2 at the client on write.
- The number of bytes read or written at the server, is returned in the data length field of the file reference variable based on the number of UCS-2 encoded characters read from or written to the file. The number of bytes actually read from or written to the file may be larger.

### **Collation**

Graphic data is sorted in binary sequence. Mixed data is sorted in the collating sequence of the database applied on each byte. For a discussion on sorting sequences, refer to the *SQL Reference*. Due to the possible difference in the ordering of characters in an EUC code set and a DBCS code set for the same country, different results may be obtained when the same data is sorted in an EUC database and in a DBCS database.

## **Developing for Mixed Code Set Environments**

This section deals with the following considerations related to the increase or decrease in the length of data under certain circumstances, when developing applications in a mixed EUC and DBCS environment:

- Unequal Code Page Situations
- Client-Based Parameter Validation
- Using the DESCRIBE Statement
- Using Fixed or Variable Length Data Types
- Character Conversion String Length Overflow
- Applications Connected to a Unicode (UCS-2) Database
- Rules for String Conversions
- Character Conversions Past Data Type Limits
- Character Conversions in Stored Procedures

## Unequal Code Page Situations

Depending on the character encoding schemes used by the application code page and the database code page, there may or may not be a change in the length of a string as it is converted from the source code page to the target code page. A change in length is usually associated with conversions between multi-byte code pages with different encoding schemes, for example DBCS and EUC.

A possible increase in length is usually more serious than a possible decrease in length since an over-allocation of memory is less problematic than an under-allocation. Application considerations for sending or retrieving data depending on where the possible expansion may occur need to be dealt with separately. It is also important to note the differences between a *best-case* and *worst-case* situation when an expansion or contraction in length is indicated. Positive values, indicating a possible expansion, will give the *worst-case* multiplying factor. For example, a value of 2 for the SQLERRD(1) or SQLERRD(2) field means that a maximum of twice the string length of storage will be required to handle the data after conversion. This is a *worst-case* indicator. In this example *best-case* would be that after conversion, the length remains the same.

Negative values for SQLERRD(1) or SQLERRD(2), indicating a possible contraction, also provide the *worst-case* expansion factor. For example, a value of -1 means that the maximum storage required is equal to the string length prior to conversion. It is indeed possible that less storage may be required, but practically this is of little use unless the receiving application knows in advance how the source data is structured.

To ensure that you always have sufficient storage allocated to cover the maximum possible expansion after character conversion, you should allocate storage equal to the value `max_target_length` obtained from the following calculation:

1. Determine the expansion factor for the data.

For data transfer from the application to the database:

```
expansion_factor = ABS[SQLERRD(1)]
if expansion_factor = 0
 expansion_factor = 1
```

For data transfer from the database to the application:

```
expansion_factor = ABS[SQLERRD(2)]
if expansion_factor = 0
 expansion_factor = 1
```

In the above calculations, ABS refers to the absolute value.

The check for `expansion_factor = 0` is necessary because some DB2 Universal Database products return 0 in `SQLERRD(1)` and `SQLERRD(2)`. These servers do not support code page conversions that result in the expansion or shrinkage of data; this is represented by an expansion factor of 1.

2. Intermediate length calculation.

```
temp_target_length = actual_source_length * expansion_factor
```

3. Determine the maximum length for target data type.

| Target data type | Maximum length of type<br>(type_maximum_length) |
|------------------|-------------------------------------------------|
| CHAR             | 254                                             |
| VARCHAR          | 32 672                                          |
| LONG VARCHAR     | 32 700                                          |
| CLOB             | 2 147 483 647                                   |

4. Determine the maximum target length.

```

1 if temp_target_length < actual_source_length
 max_target_length = type_maximum_length
else
2 if temp_target_length > type_maximum_length
 max_target_length = type_maximum_length
3 else
 max_target_length = temp_target_length

```

All the above checks are required to allow for overflow which may occur during the length calculation. The specific checks are:

- 1 Numeric overflow occurs during the calculation of `temp_target_length` in step 2.

If the result of multiplying two positive values together is greater than the maximum value for the data type, the result *wraps around* and is returned as a value less than the larger of the two values.

For example, the maximum value of a 2-byte signed integer (which is used for the length of non-CLOB data types) is 32 767. If the `actual_source_length` is 25 000 and the expansion factor is 2, then `temp_target_length` is theoretically 50 000. This value is too large for the 2-byte signed integer so it gets wrapped around and is returned as -15 536.

For the CLOB data type, a 4-byte signed integer is used for the length. The maximum value of a 4-byte signed integer is 2 147 483 647.

- 2 `temp_target_length` is too large for the data type.

The length of a data type cannot exceed the values listed in step 3.

If the conversion requires more space than is available in the data type, it may be possible to use a larger data type to hold the result. For example, if a CHAR(250) value requires 500 bytes to hold the converted string, it will not fit into a CHAR value because the maximum length is 254 bytes. However, it may be possible to use a VARCHAR(500) to hold the result after conversion. See “Character Conversions Past Data Type Limits” on page 432 for more information.

**3** temp\_target\_length is the correct length for the result.

Using the SQLERRD(1) and SQLERRD(2) values returned when connecting to the database and the above calculations, you can determine whether the length of a string will possibly increase or decrease as a result of character conversion. In general, a value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. (Note that values of ‘0’ will only come from down-level DB2 Universal Database products. Also, these values are undefined for other database server products. Table 14 lists values to expect for various application code page and database code page combinations when using DB2 Universal Database.

Table 14. SQLCA.SQLERRD Settings on CONNECT

| Application Code Page | Database Code Page | SQLERRD(1) | SQLERRD(2) |
|-----------------------|--------------------|------------|------------|
| SBCS                  | SBCS               | +1         | +1         |
| DBCS                  | DBCS               | +1         | +1         |
| eucJP                 | eucJP              | +1         | +1         |
| eucJP                 | DBCS               | -1         | +2         |
| DBCS                  | eucJP              | +2         | -1         |
| eucTW                 | eucTW              | +1         | +1         |
| eucTW                 | DBCS               | -1         | +2         |
| DBCS                  | eucTW              | +2         | -1         |
| eucKR                 | eucKR              | +1         | +1         |
| eucKR                 | DBCS               | +1         | +1         |
| DBCS                  | eucKR              | +1         | +1         |
| eucCN                 | eucCN              | +1         | +1         |
| eucCN                 | DBCS               | +1         | +1         |
| DBCS                  | eucCN              | +1         | +1         |

**Expansion at the Database Server:** If the SQLERRD(1) entry indicates an expansion at the database server, your application must consider the possibility that length-dependent character data which is valid at the client will not be valid at the database server once it is converted. For example, DB2 products require that column names be no more than 128 bytes in length. It is possible that a character string which is 128 bytes in length encoded under a DBCS code page expands past the 128 byte limit when it is converted to an EUC code page. This means that there may be activities which are valid when the application code page and the database code page are equal, which are invalid when they are different. Exercise caution when you design EUC and DBCS databases for unequal code page situations.

**Expansion at the Application:** If the SQLERRD(2) entry indicates an expansion at the client application, your application must consider the possibility that length-dependent character data will expand in length after being converted. For example, a row with a CHAR(128) column is retrieved. Under circumstances where the database and application code pages are equal, the length of the data returned is 128 bytes. However, in an unequal code page situation 128 bytes of data encoded under a DBCS code page may expand past 128 bytes when converted to an EUC code page. Thus, additional storage may have to be allocated in order to retrieve the complete string.

### **Client-Based Parameter Validation**

An important side effect of potential character data expansion or contraction between the client and server involves the validation of data passed between the client application and the database server. In an unequal code page situation, it is possible that data determined to be valid at the client is actually invalid at the database server after character conversion. Conversely, data that is invalid at the client, may be valid at the database server after conversion.

Any end-user application or API library has the potential of not being able to handle all possibilities in an unequal code page situation. In addition, while some parameter validation such as string length is performed at the client for commands and APIs, the tokens within SQL statements are not verified until they have been converted to the database's code page. This can lead to situations where it is possible to use an SQL statement in an unequal code page environment to access a database object, such as a table, but it will not be possible to access the same object using a particular command or API.

Consider an application that returns data contained in a table provided by an end-user, and checks that the table name is not greater than 128 bytes long. Now consider the following scenarios for this application:

1. A DBCS database is created. From a DBCS client, a table (t1) is created with a table name which is 128 bytes long. The table name includes several characters which would be greater than two bytes in length if the



string is converted to EUC, resulting in the EUC representation of the table name being a total of 131 bytes in length. Since there is no expansion for DBCS to DBCS connections, the table name is 128 bytes in the database environment, and the CREATE TABLE is successful.

2. An EUC client connects to the DBCS database. It creates a table (t2) with a table name which is 120 bytes long when encoded as EUC and 100 bytes long when converted to DBCS. The table name in the DBCS database is 100 bytes. The CREATE TABLE is successful.
3. The EUC client creates a table (t3) with a table name that is 64 EUC characters in length (131 bytes). When this name is converted to DBCS its length shrinks to the 128 byte limit. The CREATE TABLE is successful.
4. The EUC client invokes the application against the each of the tables (t1, t2, and t3) in the DBCS database, which results in:

| Table | Result                                                                         |
|-------|--------------------------------------------------------------------------------|
| t1    | The application considers the table name invalid because it is 131 bytes long. |
| t2    | Displays correct results                                                       |
| t3    | The application considers the table name invalid because it is 131 bytes long. |

5. The EUC client is used to query the DBCS database from the CLP. Although the table name is 131 bytes long on the client, the queries are successful because the table name is 128 bytes long at the server.

### Using the DESCRIBE Statement

A DESCRIBE performed against an EUC database will return information about mixed character and GRAPHIC columns based on the definition of these columns in the database. This information is based on code page of the server, before it is converted to the client's code page.

When you perform a DESCRIBE against a select list item which is resolved in the application context (for example VALUES SUBSTR(?,1,2)); then for any character or graphic data involved, you should evaluate the returned SQLLEN value along with the returned code page. If the returned code page is the same as the application code page, there is no expansion. If the returned code page is the same as the database code page, expansion is possible. Select list items which are FOR BIT DATA (code page 0), or in the application code page are not converted when returned to the application, therefore there is no expansion or contraction of the reported length.

**EUC Application with DBCS Database:** If your application's code page is an EUC code page, and it issues a DESCRIBE against a database with a DBCS code page, the information returned for CHAR and GRAPHIC columns is

returned in the database context. For example, a CHAR(5) column returned as part of a DESCRIBE has a value of five for the SQLLEN field. In the case of non-EUC data, you allocate five bytes of storage when you fetch the data from this column. With EUC data, this may not be the case. When the code page conversion from DBCS to EUC takes place, there may be an increase in the length of the data due to the different encoding used for characters for CHAR columns. For example, with the Traditional Chinese character set, the maximum increase is double. That is, the maximum character length in the DBCS encoding is two bytes which may increase to a maximum character length of four bytes in EUC. For the Japanese code set, the maximum increase is also double. Note, however, that while the maximum character length in Japanese DBCS is two bytes, it may increase to a maximum character length in Japanese EUC of three bytes. Although this increase appears to be only by a factor of 1.5, the single-byte Katakana characters in Japanese DBCS are only one byte in length, while they are two bytes in length in Japanese EUC. See “Character Conversion Expansion Factor” on page 409 for more information on determining the maximum size.

Possible changes in data length as a result of character conversions apply only to mixed character data. Graphic character data encoding is always the same length, two bytes, regardless of the encoding scheme. To avoid losing the data, you need to evaluate whether an unequal code page situation exists, and whether or not it is between a EUC application and a DBCS database. You can determine the database code page and the application code page from tokens in the SQLCA returned from a CONNECT statement. For more information, see “Deriving Code Page Values” on page 401, or refer to the *SQL Reference*. If such a situation exists, your application needs to allocate additional storage for mixed character data, based on the maximum expansion factor for that encoding scheme.

**DBCS Application with EUC Database:** If your application code page is a DBCS code page and issues a DESCRIBE against an EUC database, a situation similar to that in “EUC Application with DBCS Database” on page 429 occurs. However, in this case, your application may require less storage than indicated by the value of the SQLLEN field. The worst case in this situation is that all of the data is single-byte or double-byte under EUC, meaning that exactly SQLLEN bytes are required under the DBCS encoding scheme. In any other situation, less than SQLLEN bytes are required because a maximum of two bytes are required to store any EUC character.

### Using Fixed or Variable Length Data Types

Due to the possible change in length of strings when conversions occur between DBCS and EUC code pages, you should consider not using fixed length data types. Depending on whether you require blank padding, you should consider changing the SQLTYPE from a fixed length character string,

to a varying length character string after performing the DESCRIBE. For example, if an EUC to DBCS connection is informed of a maximum expansion factor of two, the application should allocate ten bytes (based on the CHAR(5) example in “EUC Application with DBCS Database” on page 429).

If the SQLTYPE is fixed-length, the EUC application will receive the column as an EUC data stream converted from the DBCS data (which itself may have up to five bytes of trailing blank pads) with further blank padding if the code page conversion does not cause the data element to grow to its maximum size. If the SQLTYPE is varying-length, the original meaning of the content of the CHAR(5) column is preserved, however, the source five bytes may have a target of between five and ten bytes. Similarly, in the case of possible data shrinkage (DBCS application and EUC database), you should consider working with varying-length data types.

An alternative to either allocating extra space or promoting the data type is to select the data in fragments. For example, to select the same VARCHAR(3000) which may be up to 6000 bytes in length after the conversion you could perform two selects, of SUBSTR(VC3000, 1, LENGTH(VC3000)/2) and SUBSTR(VC3000, (LENGTH(VC3000)/2)+1) separately into 2 VARCHAR(3000) application areas. This method is the only possible solution when the data type is no longer promotable. For example, a CLOB encoded in the Japanese DBCS code page with the maximum length of 2 gigabytes is possibly up to twice that size when encoded in the Japanese EUC code page. This means that the data will have to be broken up into fragments since there is no support for a data type in excess of 2 gigabytes in length.

### **Character Conversion String Length Overflow**

In EUC and DBCS unequal code page environments, situations may occur after conversion takes place, when there is not enough space allocated in a column to accommodate the entire string. In this case, the maximum expansion will be twice the length of the string in bytes. In cases where expansion does exceed the capacity of the column, SQLCODE -334 (SQLSTATE 22524) is returned.

This leads to situations that may not be immediately obvious or previously considered as follows:

- An SQL statement may be no longer than 32 765 bytes in length. If the statement is complex enough or uses enough constants or database object names that may be subject to expansion upon conversion, this limit may be reached earlier than expected.
- SQL identifiers are allowed to expand on conversion up to their maximum lengths which is eight bytes for short identifiers and 128 bytes for long identifiers.

- Host language identifiers are allowed to expand on conversion up to their maximum length which is 255 bytes.
- When the character fields in the SQLCA structure are converted, they are allowed to expand to no more than their maximum defined lengths.

**Rules for String Conversions:** If you are designing applications for mixed code page environments, refer to the *SQL Reference* for any of the following situations:

- Corresponding string columns in full selects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE statement
- Arguments of the scalar function COALESCE (and VALUE)
- Expression values of the IN list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In these situations, conversions may take place to the application code page instead of the database code page.

**Character Conversions Past Data Type Limits:** In EUC and DBCS unequal code page environments, situations may occur after conversion takes place, when the length of the mixed character or graphic string exceeds the maximum length allowed for that data type. If the length of the string, after expansion, exceeds the limit of the data type, then type promotion does not occur. Instead, an error message is returned indicating that the maximum allowed expansion length has been exceeded. This situation is more likely to occur while evaluating predicates than with inserts. With inserts, the column width is more readily known by the application, and the maximum expansion factor can be readily taken into account. In many cases, this side effect of character conversion can be avoided by casting the value to an associated data type with a longer maximum length. For example, the maximum length of a CHAR value is 254 bytes while the maximum length of a VARCHAR is 32672 bytes. In cases where expansion does exceed the maximum length of the data type, an SQLCODE -334 (SQLSTATE 22524) is returned.

**Character Conversions in Stored Procedures:** Mixed character or graphic data specified in host variables and SQLDAs in `sqlproc()` or `SQL CALL` invocations are converted in situations where the application and database code pages are different. In cases where string length expansion occurs as a result of conversion, you receive an SQLCODE -334 (SQLSTATE 22524) if there is not enough space allocated to handle the expansion. Thus you must be sure to provide enough space for potentially expanding strings when

developing stored procedures. You should use varying length data types with enough space allocated to allow for expansion.

## Applications Connected to a Unicode (UCS-2) Database

Note that the information contained in the previous section, “Developing for Mixed Code Set Environments” on page 424, is also applicable to a UCS-2 database.

Applications from any code page environment can connect to a Unicode database. For applications that connect to a Unicode database, the database manager converts character string data between the application code page and the database code page (UTF-8). For a UCS-2 database, GRAPHIC data is **always** in UCS-2. However, when you use the command line processor to retrieve graphic data, the graphic characters are also converted to the client code page. This conversion allows the command line processor to display graphic characters in the current font. Data loss may occur whenever the database manager converts UCS-2 characters to a client code page. Characters that the database manager cannot convert to a valid character in the client code page are replaced with the default substitution character in that code page.

When DB2 converts characters from a code page to UTF-8, the total number of bytes that represent the characters may expand or shrink, depending on the code page and the code points of the characters. 7-bit ASCII remains invariant in UTF-8, and each ASCII character requires one byte. Non-ASCII UCS-2 characters become two or three bytes each. For more information about UTF-8 conversions, see “Unicode/UCS-2 and UTF-8 Support in DB2 UDB” on page 410, or refer to the Unicode standard documents.

For applications that connect to a Unicode database, GRAPHIC data is already in Unicode. For applications that connect to DBCS databases, GRAPHIC data is converted between the application DBCS code page and the database DBCS code page. Unicode applications should perform the necessary conversions to and from Unicode themselves, or should set WCHARTYPE CONVERT option and use `wchar_t` for graphic data. For more details about this option, please see “Handling Graphic Host Variables” on page 519.

---

## Considerations for Multisite Updates

This section describes how your applications can work with remote databases and how they can work with more than one database at a time. Included in the discussion are:

- Remote Unit of Work

- Multisite Update

With DB2, you can run remote server functions such as BACKUP, RESTORE, DROP DATABASE, CREATE DATABASE and so on as if they were local applications. For more information on using these functions remotely, refer to the *Administration Guide*.

## Remote Unit of Work

A unit of work is a single logical transaction. It consists of a sequence of SQL statements in which either all of the operations are successfully performed or the sequence as a whole is considered unsuccessful.

A remote unit of work lets a user or application program read or update data at one location per unit of work. It supports access to one database within a unit of work. While an application program can access several remote databases, it can only access one database within a unit of work.

A remote unit of work has the following characteristics:

- Multiple requests per unit of work are supported.
- Multiple cursors per unit of work are supported.
- Each unit of work can access only one database.
- The application program either commits or rolls back the unit of work. In certain error circumstances, the server may roll back the unit of work.

## Multisite Update

Multisite update, also known as Distributed Unit of Work (DUOW) and Two-Phase commit, is a function that enables your applications to update data in multiple remote database servers with guaranteed integrity. A good example of a multisite update is a banking transaction that involves transfer of money from one account to another in a different database server. In such a transaction it is critical that updates that implement debit operation on one account do not get committed unless updates required to process credit to the other account are committed as well. The multisite update considerations apply when data representing these accounts is managed by two different database servers.

You can use multisite update to read and update multiple DB2 Universal Database databases within a unit of work. If you have installed DB2 Connect for OS/2, DB2 Connect for Windows NT, or DB2 Connect for AIX you can also use multisite update with other host or AS/400 application servers, such as DB2 Universal Database for OS/390 and DB2 Universal Database for AS/400. Certain restrictions apply when you use multisite update with other application servers, as described in “Multisite Update with DB2 Connect” on page 691.

A transaction manager coordinates the commit among multiple databases. If you use a transaction processing (TP) monitor environment such as CICS for AIX, the TP monitor uses its own transaction manager. Otherwise, the transaction manager supplied with DB2 is used. The DB2 Universal Database transaction manager is an XA (extended architecture) compliant resource manager. The TP monitors access data using the XA interface. Also note that the DB2 Universal Database transaction manager *is not* an XA compliant transaction manager, meaning the transaction manager can only coordinate DB2 databases.

For detailed information about multisite update, refer to the *Administration Guide*.

### **When to Use Multisite Update**

Multisite Update is most useful when you want to work with two or more databases and maintain data integrity. For example, if each branch of a bank has its own database, a money transfer application could do the following:

- Connect to the sender's database
- Read the sender's account balance and verify that enough money is present.
- Reduce the sender's account balance by the transfer amount.
- Connect to the recipient's database
- Increase the recipient's account balance by the transfer amount.
- Commit the databases.

By doing this within one unit of work, you ensure that either both databases are updated or neither database is updated.

### **Coding SQL for a Multisite Update Application**

Table 15 on page 436 illustrates how you code SQL statements for multisite update. The left column shows SQL statements that do not use multisite update; the right column shows similar statements with multisite update.



Table 15. RUOW and Multisite Update SQL Statements

| RUOW Statements                                    | Multisite Update Statements                         |
|----------------------------------------------------|-----------------------------------------------------|
| CONNECT TO D1<br>SELECT<br>UPDATE<br>COMMIT        | CONNECT TO D1<br>SELECT<br>UPDATE                   |
| CONNECT TO D2<br>INSERT<br>COMMIT                  | CONNECT TO D2<br>INSERT<br>RELEASE CURRENT          |
| CONNECT TO D1<br>SELECT<br>COMMIT<br>CONNECT RESET | SET CONNECTION D1<br>SELECT<br>RELEASE D1<br>COMMIT |

The SQL statements in the left column access only one database for each unit of work. This is a remote unit of work (RUOW) application.

The SQL statements in the right column access more than one database within a unit of work. This is a multisite update application.

Some SQL statements are coded and interpreted differently in a multisite update application:

- The current unit of work does not need to be committed or rolled back before you connect to another database.
- When you connect to another database, the current connection is not disconnected. Instead, it is put into a *dormant* state. If the CONNECT statement fails, the current connection is not affected.
- You cannot connect with the USER/USING clause if a current or dormant connection to the database already exists.
- You can use the SET CONNECTION statement to change a dormant connection to the current connection.

You can also accomplish the same thing by issuing a CONNECT statement to the dormant database. This is not allowed if you set SQLRULES to STD. You can set the value of SQLRULES using a precompiler option or the SET CLIENT command or API. The default value of SQLRULES (DB2) allows you to switch connections using the CONNECT statement.

- In a select, the cursor position is not affected if you switch to another database and then back to the original database.
- The CONNECT RESET statement does not disconnect the current connection and does not implicitly commit the current unit of work. Instead, it is equivalent to explicitly connecting to the default database (if one has been defined). If an implicit connection is not defined, SQLCODE -1024 (SQLSTATE 08003) is returned.



- You can use the RELEASE statement to mark a connection for disconnection at the next COMMIT. The RELEASE CURRENT statement applies to the current connection, the RELEASE *connection* applies to the named connection, and the RELEASE ALL statement applies to all connections.

A connection that is marked for release can still be used until it is dropped at the next COMMIT statement. A rollback does not drop the connection; this allows a retry with the connections still in place. Use the DISCONNECT statement (or precompiler option) to drop connections after a commit or rollback.

- The COMMIT statement commits all databases in the unit of work (current or dormant).
- The ROLLBACK statement rolls back all databases in the unit of work, and closes held cursors for all databases whether or not they are accessed in the unit of work.
- All connections (including dormant connections and connections marked for release) are disconnected when the application process terminates.
- Upon any successful connection (including a CONNECT statement with no options, which only queries the current connection) a number will be returned in the SQLERRD(3) and SQLERRD(4) fields of the SQLCA.

The SQLERRD(3) field returns information on whether the database connected is currently updatable in a unit of work. Its possible values are:

- 1        Updatable.
- 2        Read-only.

The SQLERRD(4) field returns the following information on the current characteristics of the connection:

- 0        Not applicable. This state is only possible if running from a down level client which uses one phase commit and is an updater.
- 1        One-phase commit.
- 2        One-phase commit (read-only). This state is only applicable to DRDA1 databases in a TP Monitor environment.
- 3        Two-phase commit.

If you are writing tools or utilities, you may want to issue a message to your users if the connection is read-only.

### **Precompiling a Multisite Update Application**

The following precompiler options are used when you precompile a multisite update application that does not use an external transaction manager as supplied in a TP Monitor environment. In a TP Monitor environment, these are ignored. Note that when you are precompiling a multisite update

application, you should set the CLP connection to a type 1 connection, otherwise you will receive an SQLCODE 30090 (SQLSTATE 25000) when you attempt to precompile your application. For information on setting the connection type, refer to the *Command Reference*. To precompile your multisite update application, specify the CONNECT 2 precompiler option:

**CONNECT (1 | 2)**

Specify CONNECT 2 to indicate that this application uses the SQL syntax for multisite update applications, as described in “Coding SQL for a Multisite Update Application” on page 435. The default, CONNECT 1, means that the normal (RUOW) rules for SQL syntax apply to the application.

**SYNCPOINT (ONEPHASE | TWOPHASE | NONE)**

Specifies whether updates can be performed on one database per unit of work (the default), multiple databases with two-phase commit (TWOPHASE), or multiple databases without two-phase commit (NONE). Note that you require that a TM\_DATABASE be defined for a TWOPHASE commit, except if you are using a transaction manager. For information on how these SYNCPOINT options impact the way your program operates, refer to the concepts section of the *SQL Reference*.

**SQLRULES (DB2 | STD)**

Specifies whether DB2 rules or standard (STD) rules based on ISO/ANSI SQL92 should be used in multisite update applications. DB2 rules allow you to issue a CONNECT statement to a dormant database; STD rules do not allow this.

**DISCONNECT (EXPLICIT | CONDITIONAL | AUTOMATIC)**

Specifies which database connections are disconnected at COMMIT: only databases that are marked for release with a RELEASE statement (EXPLICIT), all databases that have no open WITH HOLD cursors (CONDITIONAL), or all connections (AUTOMATIC).

For a more detailed description of these precompiler options, refer to the *Command Reference*.

Multisite update precompiler options become effective when the first database connection is made. You can use the SET CLIENT API to supersede connection settings when there are no existing connections (before any connection is established or after all connections are disconnected). You can use the QUERY CLIENT API to query the current connection settings of the application process.

The binder fails if an object referenced in your application program does not exist. There are three possible ways to deal with multisite update applications:

- You can split the application into several files, each of which accesses only one database. You then prep and bind each file against the one database that it accesses.
- You can ensure that each table exists in each database. For example, the branches of a bank might have databases whose tables are identical (except for the data).
- You can use only dynamic SQL.

### **Specifying Configuration Parameters for a Multisite Update Application**

The following configuration parameters are used for multisite update applications:

#### **TM\_DATABASE**

Specifies which database will act as a transaction manager for two-phase commit transactions.

#### **RESYNC\_INTERVAL**

Specifies the number of seconds that the system waits between attempts to try to resynchronize an indoubt transaction. (An indoubt transaction is a transaction that successfully completes the first phase of a two-phase commit but fails during the second phase.)

#### **LOCKTIMEOUT**

Specifies the number of seconds before a lock wait will time-out and roll back the current transaction.

#### **TP\_MON\_NAME**

Specifies the name of the TP monitor, if any.

#### **SPM\_RESYNC\_AGENT\_LIMIT**

Specifies the number of simultaneous agents that can perform resync operations with the host or AS/400 server using SNA.

#### **SPM\_NAME**

- If SPM is being used with a TCP/IP 2PC connection then the SPM\_NAME must be a unique identifier within the network. We recommend using the host name of the machine that the SPM is running on.
- If SPM is being used with an SNA 2PC connection, the SPM name must be set to the LU\_NAME that is used for 2PC.
- If SPM is being used for both TCP/IP and SNA then the LU\_NAME that is used for 2PC must be used.

**Note:** In a multisite update environment where TCP/IP is being used to connect to a host or AS/400 database server SPM may or may not be required; for more information refer to the *DB2*

*Connect Enterprise Edition for OS/2 and Windows NT Quick Beginnings or DB2 Connect Enterprise Edition for UNIX Quick Beginnings.*

#### **SPM\_LOG\_SIZE**

The number of 4 kilobyte pages of each primary and secondary log file used by the SPM to record information on connections, status of current connections, and so on.

For a more detailed description of these configuration parameters, refer to the *Administration Guide*.

#### **Multisite Update Restrictions**

The following restrictions apply to multisite update in DB2:

- In a transaction processing (TP) Monitor environment such as CICS/6000, the DISCONNECT statement is not supported. If you use DISCONNECT with a TP monitor, you will receive SQLCODE -30090 (SQLSTATE 25000). Instead of DISCONNECT, use RELEASE followed by COMMIT.
- Dynamic COMMIT and ROLLBACK are not supported in a connect type 2 environment. If you use a COMMIT in this environment, it is rejected with SQLCODE -925 (SQLSTATE 2D521). If you use a ROLLBACK in this environment, it is rejected with SQLCODE -926 (SQLSTATE 2D521).
- The precompiler option DISCONNECT CONDITIONAL cannot be used for connections to Version 1 databases. Connections to Version 1 databases are disconnected on COMMIT even if held-cursors are open.
- Although cursors declared WITH HOLD are supported with multisite update, in order for DISCONNECT to succeed, all cursors declared WITH HOLD must be closed and a COMMIT issued before the DISCONNECT request.
- When the services of TP Monitor Environments are used for transaction management, the multisite update options are implicitly CONNECT Type 2, SYNCPOINT TWOPHASE, SQLRULES DB2, DISCONNECT EXPLICIT. Changing these options with precompilation or the SET CLIENT API is not necessary and will be ignored.
- Your application receives an SQLCODE -30090 (SQLSTATE 25000) if it uses the following APIs in a multisite update (CONNECT Type 2), as these APIs are not supported in a multisite update:

BACKUP DATABASE  
BIND  
EXPORT  
IMPORT  
LOAD  
MIGRATE DATABASE  
PRECOMPILE PROGRAM

RESTART DATABASE  
RESTORE DATABASE  
REORGANIZE TABLE  
ROLLFORWARD DATABASE

- Stored procedures are supported within a multisite update. However, a stored procedure that issues a COMMIT and ROLLBACK statement in a multisite update (CONNECT Type 2) receives an SQLCODE -30090 (SQLSTATE 25000) as these statements are not supported in a multisite update.

---

## Accessing Host or AS/400 Servers

If you want to develop applications that can access (or update) different database systems, you should:

1. Use SQL statements and precompile/bind options that are supported on all of the database systems that your applications will access. For example, stored procedures are not supported on all platforms.

For IBM products, refer to the *SQL Reference* **before** you start coding.

2. Where possible, have your applications check the SQLSTATE rather than the SQLCODE.

If your applications will use DB2 Connect and you want to use SQLCODEs, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.

3. Test your application with the host or AS/400 databases (such as DB2 Universal Database for OS/390, OS/400, or DB2 for VSE & VM) that you intend to support. For more information, refer to the *DB2 Connect User's Guide*.

For more information on accessing host or AS/400 database systems, see "Appendix D. Programming in a Host or AS/400 Environment" on page 679.

---

## Multiple Thread Database Access

One feature of some operating systems is the ability to run several threads of execution within a single process. This allows an application to handle asynchronous events, and makes it easier to create event-driven applications without resorting to polling schemes. This section discusses how the database manager works with multiple threads, and lists some design guidelines that you should keep in mind. To determine if your platform supports the multithreading feature, refer to the *Application Building Guide*.

This section assumes that you are familiar with the terms relating to the development of multithreaded applications (such as critical section and

semaphore). If you are not familiar with these terms, consult the programming documentation for your operating system.

A DB2 application can execute SQL statements from multiple threads using *contexts*. A context is the environment from which an application runs all SQL statements and API calls. All connections, units of work, and other database resources are associated with a specific context. Each context is associated with one or more threads within an application.

For each executable SQL statement in a context, the first run-time services call always tries to obtain a latch. If it is successful, it continues processing. If not (because an SQL statement in another thread of the same context already has the latch), the call is blocked on a signaling semaphore until that semaphore is posted, at which point the call gets the latch and continues processing. The latch is held until the SQL statement has completed processing, at which time it is released by the last run-time services call that was generated for that particular SQL statement.

The net result is that each SQL statement within a context is executed as an atomic unit, even though other threads may also be trying to execute SQL statements at the same time. This action ensures that internal data structures are not altered by different threads at the same time. APIs also use the latch used by run-time services; therefore, APIs have the same restrictions as run-time services routines within each context.

By default, all applications have a single context that is used for all database access. While this is perfect for a single threaded application, the serialization of SQL statements makes a single context inadequate for a multithreaded application. By using the following DB2 APIs, your application can attach a separate context to each thread and allow contexts to be passed between threads:

- `sqlcSetTypeCtx()`
- `sqlcBeginCtx()`
- `sqlcEndCtx()`
- `sqlcAttachToCtx()`
- `sqlcDetachFromCtx()`
- `sqlcGetCurrentCtx()`
- `sqlcInterruptCtx()`

Contexts may be exchanged between threads in a process, but not exchanged between processes. One use of multiple contexts is to provide support for concurrent transactions. For the details of how to use these context APIs, refer to the *Administrative API Reference* and “Concurrent Transactions” on page 446.

## Recommendations for Using Multiple Threads

Follow these guidelines when accessing a database from multiple thread applications:

- **Serialize alteration of data structures.**

Applications must ensure that user-defined data structures used by SQL statements and database manager routines are not altered by one thread while an SQL statement or database manager routine is being processed in another thread. For example, do not allow a thread to reallocate an SQLDA while it was being used by an SQL statement in another thread.

- **Consider using separate data structures.**

It may be easier to give each thread its own user-defined data structures to avoid having to serialize their usage. This is especially true for the SQLCA, which is used not only by every executable SQL statement, but also by all of the database manager routines. There are three alternatives for avoiding this problem with the SQLCA:

1. Use EXEC SQL INCLUDE SQLCA, but add `struct sqlca sqlca` at the beginning of any routine which is used by any thread other than the first thread.
2. Place EXEC SQL INCLUDE SQLCA inside each routine that contains SQL, instead of in the global scope.
3. Replace EXEC SQL INCLUDE SQLCA with `#include "sqlca.h"` and then add `"struct sqlca sqlca"` at the beginning of any routine that uses SQL.

## Multithreaded UNIX Applications Working with Code Page and Country Code

On AIX, Solaris, HP-UX, SCO UnixWare 7, and Silicon Graphics IRIX, changes have been made to the functions that are used for run time querying of the code page and country code to be used for a database connection. They are now thread safe but can create some lock contention (and resulting performance degradation) in a multithreaded application which uses a large number of concurrent database connections.

A new environment variable has been created (DB2\_FORCE\_NLS\_CACHE) to eliminate the chance of lock contention in multithreaded applications. When DB2\_FORCE\_NLS\_CACHE is set to TRUE the code page and country code information is saved the first time a thread accesses it. From that point on the cached information will be used for any other thread that requests this information. By saving this information, lock contention is eliminated and in certain situations a performance benefit will be realized.

DB2\_FORCE\_NLS\_CACHE should not be set to true if the application changes locale settings between connections. If this is done then the original locale information will be returned even after the locale settings have been changed. In general, multithreaded applications will not change locale settings. This ensures that the application remains thread safe.

## Potential Pitfalls when Using Multiple Threads

An application that uses multiple threads is, understandably, more complex than a single-threaded application. This extra complexity can potentially lead to some unexpected problems. When writing a multithreaded application, exercise caution with the following:

- **Database dependencies between two or more contexts.**

Each context in an application has its own set of database resources, including locks on database objects. This makes it possible for two contexts, if they are accessing the same database object, to deadlock. The database manager will detect the deadlock and one of the contexts will receive SQLCODE -911 and its unit of work will be rolled back.

- **Application dependencies between two or more contexts.**

Be careful with any programming techniques that establish inter-context dependencies. Latches, semaphores, and critical sections are examples of programming techniques that can establish such dependencies. If an application has two contexts that have both application and database dependencies between the contexts, it is possible for the application to become deadlocked. If some of the dependencies are outside of the database manager, the deadlock is not detected, thus the application gets suspended or hung.

As an example of this sort of problem, consider an application that has two contexts, both of which access a common data structure. To avoid problems where both contexts change the data structure simultaneously, the data structure is protected by a semaphore. The contexts look like this:

```
context 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
get semaphore
access data structure
release semaphore
COMMIT
```

```
context 2
get semaphore
access data structure
SELECT * FROM TAB1...
release semaphore
COMMIT
```



Suppose the first context successfully executes the SELECT and the UPDATE statements while the second context gets the semaphore and accesses the data structure. The first context now tries to get the semaphore, but it cannot because the second context is holding the semaphore. The second context now attempts to read a row from table TAB1, but it stops on a database lock held by the first context. The application is now in a state where context 1 cannot finish before context 2 is done and context 2 is waiting for context 1 to finish. The application is deadlocked, but because the database manager does not know about the semaphore dependency neither context will be rolled back. This leaves the application suspended.

### Preventing Deadlocks for Multiple Contexts

Because the database manager cannot detect deadlocks between threads, design and code your application in a way that will prevent deadlocks (or at least allow them to be avoided). In the above example, you can avoid the deadlock in several ways:

- Release all locks held before obtaining the semaphore.  
Change the code for context 1 to perform a commit before it gets the semaphore.
- Do not code SQL statements inside a section protected by semaphores.  
Change the code for context 2 to release the semaphore before doing the SELECT.
- Code all SQL statements within semaphores.  
Change the code for context 1 to obtain the semaphore before running the SELECT statement. While this technique will work, it is not highly recommended because the semaphores will serialize access to the database manager, which potentially negates the benefits of using multiple threads.
- Set the LOCKTIMEOUT database configuration parameter to a value other than -1.  
While this will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. When handling the roll back error, context 2 should release the semaphore. Once the semaphore has been released, context 1 can continue and context 2 is free to retry its work .

The techniques for avoiding deadlocks are shown in terms of the above example, but you can apply them to all multithreaded applications. In general, treat the database manager as you would treat any protected resource and you should not run into problems with multithreaded applications.

---

## Concurrent Transactions

Sometimes it is useful for an application to have multiple independent connections called *concurrent transactions*. Using concurrent transactions, an application can connect to several databases at the same time, and can establish several distinct connections to the same database.

The context APIs described in “Multiple Thread Database Access” on page 441 allow an application to use concurrent transactions. Each context created in an application is independent from the other contexts. This means you create a context, connect to a database using the context, and run SQL statements against the database without being affected by the activities such as running COMMIT or ROLLBACK statements of other contexts.

For example, suppose you are creating an application that allows a user to run SQL statements against one database, and keeps a log of the activities performed in a second database. Since the log must be kept up to date, it is necessary to issue a COMMIT statement after each update of the log, but you do not want the user’s SQL statements affected by commits for the log. This is a perfect situation for concurrent transactions. In your application, create two contexts: one connects to the user’s database and is used for all the user’s SQL; the other connects to the log database and is used for updating the log. With this design, when you commit a change to the log database, you do not affect the user’s current unit of work.

Another benefit of concurrent transactions is that if the work on the cursors in one connection is rolled back, it has no affect on the cursors in other connections. After the rollback in the one connection, both the work done and the cursor positions are still maintained in the other connections.

### Potential Pitfalls when Using Concurrent Transactions

An application that uses concurrent transactions can encounter some problems that cannot arise when writing an application that uses a single connection. When writing an application with concurrent transactions, exercise caution with the following:

- Database dependencies between two or more contexts.  
Each context in an application has its own set of database resources, including locks on database objects. This makes it possible for two contexts, if they are accessing the same database object, to become deadlocked. The database manager will detect the deadlock and one of the contexts will receive an SQLCODE -911 and its unit of work will be rolled back.
- Application dependencies between two or more contexts.

Switching contexts within a single thread creates dependencies between the contexts. If the contexts also have database dependencies, it is possible for a deadlock to develop. Since some of the dependencies are outside of the database manager, the deadlock will not be detected and the application will be suspended.

As an example of this sort of problem, consider the following application:

```
context 1
UPDATE TAB1 SET COL = :new_val

context 2
SELECT * FROM TAB1
COMMIT

context 1
COMMIT
```

Suppose the first context successfully executes the UPDATE statement. The update establishes locks on all the rows of TAB1. Now context 2 tries to select all the rows from TAB1. Since the two contexts are independent, context 2 waits on the locks held by context 1. Context 1, however, cannot release its locks until context 2 finishes executing. The application is now deadlocked, but the database manager does not know that context 1 is waiting on context 2 so it will not force one of the contexts to be rolled back. This leaves the application suspended.

### Preventing Deadlocks for Concurrent Transactions

Because the database manager cannot detect deadlocks between contexts, you must design and code your application in a way that will prevent deadlocks (or at least avoids deadlocks). In the above example, you can avoid the deadlock in several ways:

- Release all locks held before switching contexts.  
Change the code so that context 1 performs its commit before switching to context 2.
- Do not access a given object from more than one context at a time.  
Change the code so that both the update and the select are done from the same context.
- Set the LOCKTIMEOUT database configuration parameter to a value other than -1.

While this will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. Once context 2 is rolled back, context 1 can continue executing (which releases the locks) and context 2 can retry its work.

The techniques for avoiding deadlocks are shown in terms of the above example, but you can apply them to all applications which use concurrent transactions.

---

## X/Open XA Interface Programming Considerations

The X/Open® XA Interface is an open standard for coordinating changes to multiple resources, while ensuring the integrity of these changes. Software products known as *transaction processing monitors* typically use the XA interface, and since DB2 supports this interface, one or more DB2 databases may be concurrently accessed as resources in such an environment. For information about the concepts and implementation of the XA interface support provided by the database manager, refer to the *Administration Guide, Design and Implementation*. To determine if your platform supports the X/Open XA Interface, refer to the *Application Building Guide*.

Special consideration is required by DB2 when operating in a Distributed Transaction Processing (DTP) environment which uses the XA interface because a different model is used for transaction processing as compared to applications running independent of a TP monitor. The characteristics of this transaction processing model are:

1. Multiple types of recoverable resources (such as DB2 databases) can be modified within a transaction.
2. Resources are updated using two-phase commit to ensure the integrity of the transactions being executed.
3. Application programs send requests to commit or rollback a transaction to the TP monitor product rather than to the managers of the resources. For example, in a CICS environment an application would issue EXEC CICS SYNCPOINT to commit a transaction, and issuing EXEC SQL COMMIT to DB2 would be invalid and unnecessary.
4. Authorization to run transactions is screened by the TP monitor and related software, so resource managers such as DB2 treat the TP monitor as the single authorized user. For example, any use of a CICS transaction must be authenticated by CICS and the access privilege to the database must be granted to CICS and not to the end user who invokes the CICS application.
5. Multiple programs (transactions) are typically queued and executed on an application server (which appears to DB2 to be a single, long-running application program).

Due to the unique nature of this environment, DB2 has special behavior and requirements for applications coded to run in it:

- Multiple databases can be connected to and updated within a unit of work without consideration of distributed unit of work precompiler options or client settings.
- The DISCONNECT statement is disallowed, and will be rejected with SQLCODE -30090 (SQLSTATE 25000) if attempted.
- The RELEASE statement can be used to specify databases connections to release when a transaction is committed, but this is not recommended. If a connection has been released, subsequent transactions should use the SET CONNECTION statement to connect to the database without requiring authorization.
- COMMIT and ROLLBACK statements are not allowed within stored procedures accessed by a TP monitor transaction.
- When two-phase commit flows are explicitly disabled for a transaction (these are called *LOCAL* transactions in XA Interface terminology) only one database can be accessed within that transaction. This database cannot be one that is accessed using the DB2 Sync Point Manager for two-phase commit.
- LOCAL transactions should issue SQL COMMIT or SQL ROLLBACK at the end of each transaction, otherwise the transaction will be considered part of the next transaction which is processed.
- Switching between current database connections is done through the use of either SQL CONNECT or SQL SET CONNECTION. The authorization used for a connection cannot be changed by specifying a userid or password on the CONNECT statement.
- If a database object such as a table, view, or index is not fully qualified in a dynamic SQL statement, it will be implicitly qualified with the single authentication ID that the TP monitor is executing under, rather than user's ID.
- Any use of DB2 COMMIT or ROLLBACK statements for transactions that are not LOCAL will be rejected. The following codes will be returned:
  - SQLCODE -925 (SQLSTATE 2D521) for static COMMIT
  - SQLCODE -926 (SQLSTATE 2D521) for static ROLLBACK
  - SQLCODE -426 (SQLSTATE 2D528) for dynamic COMMIT
  - SQLCODE -427 (SQLSTATE 2D529) for dynamic ROLLBACK
- CLI requests to COMMIT or ROLLBACK are also rejected.
- Handling database-initiated rollback

In a DTP environment, if an RM has initiated a rollback (for instance, due to a system error or deadlock) to terminate its own branch of a global transaction, it must not process any more requests from the same application process until a transaction manager-initiated syncpoint request occurs. This includes deadlocks that occur within a stored procedure. For the database manager, this means rejecting all subsequent SQL requests

with SQLCODE -918 (SQLSTATE 51021) to inform you that you must roll back the global transaction with the transaction manager's syncpoint service such as using the CICS SYNCPOINT ROLLBACK command in a CICS environment. If for some reason you request the TM to commit the transaction instead, the RM will inform the TM about the rollback and cause the TM to roll back other RMs anyway.

- **Cursors declared WITH HOLD**

Cursors declared WITH HOLD are supported in XA/DTP environments for CICS transaction processing monitors.

In cases where cursors declared WITH HOLD are not supported, the OPEN statement will be rejected with SQLCODE -30090 (SQLSTATE 25000), reason code 03.

It is the responsibility of the transactions to ensure that cursors specified to be WITH HOLD are explicitly closed when they are no longer required; otherwise they might be inherited by other transactions, causing conflict or unnecessary use of resources.

- Statements which update or change a database are not allowed against databases which do not support two-phase commit request flows. For example, accessing host or AS/400 database servers in environments in which level 2 of DRDA protocol (DRDA2) is not supportable (see "Multisite Update with DB2 Connect" on page 691).
- Whether a database supports updates in an XA environment can be determined at run-time by issuing a CONNECT statement. The third SQLERRD token will have the value 1 if the database is updatable, and otherwise will have the value 2.
- When updates are restricted, only the following SQL statements will be allowed:

```
CONNECT
DECLARE
DESCRIBE
EXECUTE IMMEDIATE (where the first token or keyword is SET but
 not SET CONSTRAINTS)

OPEN CURSOR
FETCH CURSOR
CLOSE CURSOR
PREPARE (where the first token or keyword that is not blank or
 left parenthesis is SET (other than SET CONSTRAINTS),
 SELECT, WITH, or VALUES)
SELECT...INTO
VALUES...INTO
```

Any other attempts will be rejected with SQLCODE -30090 (SQLSTATE 25000).

The PREPARE statement will only be useable to prepare SELECT statements. The EXECUTE IMMEDIATE statement is also allowed to

execute SQL SET statements that do not return any output value, such as the SET SQLID statement from DB2 Universal Database for OS/390.

- API Restrictions

APIs which internally issue a commit in the database and bypass the two-phase commit process will be rejected with SQLCODE -30090 (SQLSTATE 25000). For a list of these APIs, see “Multisite Update Restrictions” on page 440. These APIs are not supported in a multisite update (Connect Type 2).

- Applications should be single-threaded.

If you intend to develop a multithreaded application, you should ensure that only one thread uses SQL, or use a multiprocess design instead to avoid interleaving of SQL statements from different threads within the same unit of work. If a transaction manager supports multiple processes or multithreading, you should configure it to serialize the threads so that one thread will execute to a syncpoint before another one begins. An example is the **XASerialize** option of **all\_operation** in AIX/CICS. For more details about the AIX/CICS XAD file which contains this information, refer to the *Administration Guide, Design and Implementation*.

Note that the above restrictions apply to applications running in TP monitor environment which uses the XA interface. If DB2 databases are not defined for use with the XA interface, these restrictions do not apply, however it is still necessary to ensure that transactions are coded in a way that will not leave DB2 in a state which will adversely affect the next transaction to be run.

## Application Linkage

To produce an executable application, you need to link in the application objects with the language libraries, the operating system libraries, the normal database manager libraries, and the libraries of the TP monitor and transaction manager products.

If you are writing applications on a CICS for AIX® system, refer to the *CICS for AIX Application Programming Guide*, SC33-0814, for SQL restrictions and sample CICS SQL programs.

---

## Working with Large Volumes of Data Across a Network

You can combine the techniques of stored procedures, described in “Chapter 6. Stored Procedures” on page 185, and row blocking, described in the *Administration Guide, Performance*, to significantly improve the performance of applications which need to pass large amounts of data across a network.

Applications that pass arrays, large amounts of data, or packages of data across the network can pass the data in blocks using the SQLDA data structure or host variables as the transport mechanism. This technique is extremely powerful in host languages that support structures.

Either a client application or a server procedure can pass the data across the network. It can be passed using one of the following data types:

- VARCHAR
- LONG VARCHAR
- CLOB
- BLOB

It can also be passed using one of the following graphic types:

- VARGRAPHIC
- LONG VARGRAPHIC
- DBCLOB

See “Data Types” on page 75 for more information about this topic.

**Note:** Be sure to consider the possibility of character conversion when using this technique. If you are passing data with one of the character string data types such as VARCHAR, LONG VARCHAR, or CLOB, or graphic data types such as VARGRAPHIC, LONG VARGRAPHIC, OR DBCLOB, and the application code page is not the same as the database code page, any non-character data will be converted as if it were character data. To avoid character conversion, you should pass data in a variable with a data type of BLOB.

See “Conversion Between Different Code Pages” on page 407 for more information about how and when data conversion occurs.



---

## Chapter 14. Programming Considerations in a Partitioned Environment

|                                                  |     |                                            |     |
|--------------------------------------------------|-----|--------------------------------------------|-----|
| Improving Performance . . . . .                  | 453 | Creating a Test Environment . . . . .      | 466 |
| Using FOR READ ONLY Cursors . . . . .            | 453 | Error-Handling Considerations . . . . .    | 467 |
| Using Directed DSS and Local Bypass . . . . .    | 453 | Severe Errors . . . . .                    | 467 |
| Directed DSS . . . . .                           | 454 | Merged Multiple SQLCA Structures . . . . . | 468 |
| Using Local Bypass . . . . .                     | 454 | Identifying the Partition that Returned    |     |
| Using Buffered Inserts . . . . .                 | 455 | the Error . . . . .                        | 469 |
| Considerations for Using Buffered                |     | Debugging . . . . .                        | 469 |
| Inserts . . . . .                                | 458 | Diagnosing a Looping or Suspended          |     |
| Restrictions on Using Buffered Inserts . . . . . | 460 | application . . . . .                      | 469 |
| Example: Extracting Large Volume of              |     |                                            |     |
| Data (largevol.c) . . . . .                      | 461 |                                            |     |

---

### Improving Performance

To take advantage of the performance benefits that partitioned environments offer, you should consider using special programming techniques. For example, if your application accesses DB2 data from more than one database manager partition, you need to consider the information contained herein. For an overview of partitioned environments, refer to the *Administration Guide* and the *SQL Reference*.

#### Using FOR READ ONLY Cursors

If you declare a cursor from which you intend only to read, include FOR READ ONLY or FOR FETCH only in the OPEN CURSOR declaration. (FOR READ ONLY and FOR FETCH ONLY are equivalent statements.) FOR READ ONLY cursors allow the coordinator partition to retrieve multiple rows at a time, dramatically improving the performance of subsequent FETCH statements. When you do not explicitly declare cursors FOR READ ONLY, the coordinator partition treats them as updatable cursors. Updatable cursors incur considerable expense because they require the coordinator partition to retrieve only a single row per FETCH.

#### Using Directed DSS and Local Bypass

To optimize Online Transaction Processing applications, you may want to avoid simple SQL statements that require processing on all data partitions. You should design the application so that SQL statements can retrieve data from single partitions. These techniques avoid the expense the coordinator partition incurs communicating with one or all of the associated partitions.

## Directed DSS

A *directed DSS* uses the table partition key to direct a query to a single partition. Use this type of query in your application to avoid the coordinator partition overhead required for a query broadcast to all nodes.

An example SELECT statement fragment that can take advantage of directed DSS follows:

```
SELECT ... FROM t1
WHERE PARTKEY=:hostvar
```

When the coordinator partition receives the query, it determines which partition holds the subset of data for *:hostvar* and directs the query specifically to that partition.

To optimize your application using directed DSS, divide complex queries into multiple simple queries. For example, in the following query the coordinator partition matches the partition key with multiple values. Because the data that satisfies the query lies on multiple partitions, the coordinator partition broadcasts the query to all partitions:

```
SELECT ... FROM t1
WHERE PARTKEY IN (:hostvar1, :hostvar2)
```

Instead, break the query into multiple SELECT statements (each with a single host variable) or use a single SELECT statement with a UNION to achieve the same result. The coordinator partition can take advantage of simpler SELECT statements to use directed DSS to communicate only to the necessary partitions. The optimized query looks like:

```
SELECT ... AS res1 FROM t1
WHERE PARTKEY=:hostvar1
UNION
SELECT ... AS res2 FROM t1
WHERE PARTKEY=:hostvar2
```

Note that the above technique will only improve performance if the number of selects in the UNION is significantly smaller than the number of partitions.

## Using Local Bypass

A specialized form of the directed DSS query accesses data stored only on the coordinator partition. This is called a *local bypass* because the coordinator partition completes the query without having to communicate with another partition.

Local bypass is enabled automatically whenever possible, but you can increase its use by routing transactions to the partition containing the data for that transactions. One technique for doing this is to have a remote client maintain

connections to each partition. A transaction can then use the correct connection based on the input partition key. Another technique is to group transaction by partition and have separate application server for each partition.

In order to determine the number of the partition on which transaction data resides, you can use the `sqlugrpn` API (Get Row Partitioning Number). This API allows an application to efficiently calculate the partition number of a row, given the partitioning key. For more information on the `sqlugrpn` API, refer to the *Administrative API Reference*.

Another alternative is to use the `db2atld` utility to divide input data by partition number and run a copy of the application against each partition. For more information on the `db2atld` utility, refer to the *Command Reference*.

## Using Buffered Inserts

A buffered insert is an insert statement that takes advantage of table queues to buffer the rows being inserted, thereby gaining a significant performance improvement. To use a buffered insert, an application must be prepared or bound with the `INSERT BUF` option.

Buffered inserts can result in substantial performance improvement in applications that perform inserts. Typically, you can use a buffered insert in applications where a single insert statement (and no other database modification statement) is used within a loop to insert many rows and where the source of the data is a `VALUES` clause in the `INSERT` statement. Typically the `INSERT` statement is referencing one or more host variables which change their values during successive executions of the loop. The `VALUES` clause can specify a single row or multiple rows.

Typical decision support applications require the loading and periodic insertion of new data. This data could be hundreds of thousands of rows. You can prepare and bind applications to use buffered inserts when loading tables.

To cause an application to use buffered inserts, use the `PREP` command to process the application program source file, or use the `BIND` command on the resulting bind file. In both situations, you must specify the `INSERT BUF` option. For more information about binding an application, see “Binding” on page 51. For more information about preparing an application, see “Creating and Preparing the Source Files” on page 45.

**Note:** Buffered inserts cause the following steps to occur:

1. The database manager opens one 4 KB buffer for each node on which the table resides.

2. The INSERT statement with the VALUES clause issued by the application causes the row (or rows) to be placed into the appropriate buffer (or buffers).
3. The database manager returns control to the application.
4. The rows in the buffer are sent to the partition when the buffer becomes full, or an event occurs that causes the rows in a partially filled buffer to be sent. A partially filled buffer is flushed when one of the following occurs:
  - The application issues a COMMIT (implicitly or explicitly through application termination) or ROLLBACK.
  - The application issues another statement that causes a savepoint to be taken. OPEN, FETCH, and CLOSE cursor statements do not cause a savepoint to be taken, nor do they close an open buffered insert.

The following SQL statements will close an open buffered insert:

- BEGIN COMPOUND SQL
- COMMIT
- DDL
- DELETE
- END COMPOUND SQL
- EXECUTE IMMEDIATE
- GRANT
- INSERT to a different table
- PREPARE of the same dynamic statement (by name) doing buffered inserts
- REDISTRIBUTE NODEGROUP
- REORG
- REVOKE
- ROLLBACK
- RUNSTATS
- SELECT INTO
- UPDATE
- Execution of any other statement, but not another (looping) execution of the buffered INSERT
- End of application

The following APIs will close an open buffered insert:

- BIND (API)
- REBIND (API)
- RUNSTATS (API)

- REORG (API)
- REDISTRIBUTE (API)

In any of these situations where another statement closes the buffered insert, the coordinator node waits until every node receives the buffers and the rows are inserted. It then executes the other statement (the one closing the buffered insert), provided all the rows were successfully inserted. See “Considerations for Using Buffered Inserts” on page 458 for additional details.

The standard interface in a partitioned environment, (without a buffered insert) loads one row at a time doing the following steps (assuming that the application is running locally on one of the partitions):

1. The coordinator node passes the row to the database manager that is on the same node.
2. The database manager uses indirect hashing to determine the partition where the row should be placed:
  - The target partition receives the row.
  - The target partition inserts the row locally.
  - The target partition sends a response to the coordinator node.
3. The coordinator node receives the response from the target partition.
4. The coordinator node gives the response to the application  
The insertion is not committed until the application issues a COMMIT.
5. Any INSERT statement containing the VALUES clause is a candidate for Buffered Insert, regardless of the number of rows or the type of elements in the rows. That is, the elements can be constants, special registers, host variables, expressions, functions and so on.

For a given INSERT statement with the VALUES clause, the DB2 SQL compiler may not buffer the insert based on semantic, performance, or implementation considerations. If you prepare or bind your application with the INSERT BUF option, ensure that it is not dependent on a buffered insert. This means:

- Errors may be reported asynchronously for buffered inserts, or synchronously for regular inserts. If reported asynchronously, an insert error may be reported on a subsequent insert within the buffer, or on the *other* statement which closes the buffer. The statement that reports the error is not executed. For example, consider using a COMMIT statement to close a buffered insert loop. The commit reports an SQLCODE -803 (SQLSTATE 23505) due to a duplicate key from an earlier insert. In this scenario, the commit is not executed. If you want your application to really commit, for example, some updates that are performed before it enters the buffered insert loop, you must reissue the COMMIT statement.

- Rows inserted may be immediately visible through a SELECT statement using a cursor without a buffered insert. With a buffered insert, the rows will not be immediately visible. Do not write your application to depend on these cursor-selected rows if you precompile or bind it with the INSERT BUF option.

Buffered inserts result in the following performance advantages:

- Only one message is sent from the target partition to the coordinator node for each buffer received by the target partition.
- A buffer can contain a large number of rows, especially if the rows are small.
- Parallel processing occurs as insertions are being done across partitions while the coordinator node is receiving new rows.

An application that is bound with INSERT BUF should be written so that the same INSERT statement with VALUES clause is iterated repeatedly before any statement or API that closes a buffered insert is issued.

**Note:** You should do periodic commits to prevent the buffered inserts from filling the transaction log.

### Considerations for Using Buffered Inserts

Buffered inserts exhibit behaviors that can affect an application program. This behavior is caused by the asynchronous nature of the buffered inserts. Based on the values of the row's partitioning key, each inserted row is placed in a buffer destined for the correct partition. These buffers are sent to their destination partitions as they become full, or an event causes them to be flushed. You must be aware of the following, and account for them when designing and coding the application:

- Certain error conditions for inserted rows are not reported when the INSERT statement is executed. They are reported later, when the first statement other than the INSERT (or INSERT to a different table) is executed, such as DELETE, UPDATE, COMMIT, or ROLLBACK. Any statement or API that closes the buffered insert statement can see the error report. Also, any invocation of the insert itself may see an error of a previously inserted row. Moreover, if a buffered insert error is reported by another statement, such as UPDATE or COMMIT, DB2 will not attempt to execute that statement.
- An error detected during the insertion of a *group of rows* causes all the rows of that group to be backed out. A group of rows is defined as all the rows inserted through executions of a buffered insert statement:
  - From the beginning of the unit of work,
  - Since the statement was prepared (if it is dynamic), or

- Since the previous execution of another updating statement. For a list of statements that close (or flush) a buffered insert, see “Using Buffered Inserts” on page 455.
- An inserted row may not be immediately visible to SELECT statements issued after the INSERT by the same application program, if the SELECT is executed using a cursor.

A buffered INSERT statement is either open or closed. The first invocation of the statement opens the buffered INSERT, the row is added to the appropriate buffer, and control is returned to the application. Subsequent invocations add rows to the buffer, leaving the statement open. While the statement is open, buffers may be sent to their destination partitions, where the rows are inserted into the target table's partition. If any statement or API that closes a buffered insert is invoked while a buffered INSERT statement is open (including invocation of a *different* buffered INSERT statement), or if a PREPARE statement is issued against an open buffered INSERT statement, the open statement is closed before the new request is processed. If the buffered INSERT statement is closed, the remaining buffers are flushed. The rows are then sent to the target partitions and inserted. Only after all the buffers are sent and all the rows are inserted does the new request begin processing.

If errors are detected during the closing of the INSERT statement, the SQLCA for the new request will be filled in describing the error, and the new request is not done. Also, the entire group of rows that were inserted through the buffered INSERT statement *since it was opened* are removed from the database. The state of the application will be as defined for the particular error detected. For example:

- If the error is a deadlock, the transaction is rolled back (including any changes made before the buffered insert section was opened).
- If the error is a unique key violation, the state of the database is the same as before the statement was opened. The transaction remains active, and any changes made before the statement was opened are not affected.

For example, consider the following application that is bound with the buffered insert option:

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
 READ VALUE OF hv1 FROM A FILE;
 EXEC SQL INSERT INTO t2 VALUES (:hv1);
 IF 1000 INSERTS DONE, THEN DO
 EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
 RESET COUNTER;
 END;
END;
EXEC SQL COMMIT;
```

Suppose the file contains 8 000 values, but value 3 258 is not legal (for example, a unique key violation). Each 1 000 inserts results in the execution of another SQL statement, which then closes the INSERT INTO t2 statement. During the fourth group of 1 000 inserts, the error for value 3 258 will be detected. It may be detected after the insertion of more values (not necessarily the next one). In this situation, an error code is returned for the INSERT INTO t2 statement.

The error may also be detected when an insertion is attempted on table t3, which closes the INSERT INTO t2 statement. In this situation, the error code is returned for the INSERT INTO t3 statement, even though the error applies to table t2.

Suppose, instead, that you have 3 900 rows to insert. Before being told of the error on row number 3 258, the application may exit the loop and attempt to issue a COMMIT. The unique-key-violation return code will be issued for the COMMIT statement, and the COMMIT will not be performed. If the application wants to COMMIT the 3000 rows which are in the database thus far (the last execution of EXEC SQL INSERT INTO t3 ... ends the savepoint for those 3000 rows), then the COMMIT has to be REISSUED! Similar considerations apply to ROLLBACK as well.

**Note:** When using buffered inserts, you should carefully monitor the SQLCODES returned to avoid having the table in an indeterminate state. For example, if you remove the SQLCODE < 0 clause from the THEN DO statement in the above example, the table could end up containing an indeterminate number of rows.

### Restrictions on Using Buffered Inserts

The following restrictions apply:

- For an application to take advantage of the buffered inserts, one of the following must be true:
  - The application must either be prepared through PREP or bound with the BIND command and the INSERT BUF option is specified.
  - The application must be bound using the BIND or the PREP API with the SQL\_INSERT\_BUF option.
- If the INSERT statement with VALUES clause includes long fields or LOBS in the explicit or implicit column list, the INSERT BUF option is ignored for that statement and a normal insert section is done, not a buffered insert. This is not an error condition, and no error or warning message is issued.
- INSERT with fullselect is not affected by INSERT BUF. A buffered INSERT does not improve the performance of this type of INSERT.



- Buffered inserts can be used only in applications, and not through CLP-issued inserts, as these are done through the EXECUTE IMMEDIATE statement.

The application can then be run from any supported client platform.

### **Example: Extracting Large Volume of Data (largevol.c)**

Although DB2 Universal Database provides excellent features for parallel query processing, the single point of connection of an application or an EXPORT command can become a bottleneck if you are extracting large volumes of data. This occurs because the passing of data from the database manager to the application is a CPU-intensive process that executes on a single node (typically a single processor as well).

DB2 Universal Database provides several methods to overcome the bottleneck, so that the volume of extracted data scales linearly per unit of time with an increasing number of processors. The following example describes the basic idea behind these methods.

Assume that you have a table called EMPLOYEE which is stored on 20 nodes, and you generate a mailing list (FIRSTNME, LASTNAME, JOB) of all employees who are in a legitimate department (that is, WORKDEPT is not NULL).

The following query is run on each node in parallel, and then generates the entire answer set at a single node (the coordinator node):

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
```

But, the following query could be run on each partition in the database (that is, if there are five partitions, five separate queries are required, one at each partition). Each query generates the set of all the employee names whose record is on the particular partition where the query runs. Each local result set can be redirected to a file. The result sets then need to be merged into a single result set.

On AIX, you can use a property of Network File System (NFS) files to automate the merge. If all the partitions direct their answer sets to the same file on an NFS mount, the results are merged. Note that using NFS without blocking the answer into large buffers results in very poor performance.

```
SELECT FIRSTNME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
AND NODENUMBER(NAME) = CURRENT NODE
```

The result can either be stored in a local file (meaning that the final result would be 20 files, each containing a portion of the complete answer set), or in a single NFS-mounted file.

The following example uses the second method, so that the result is in a single file that is NFS mounted across the 20 nodes. The NFS locking mechanism ensures serialization of writes into the result file from the different partitions. Note that this example, as presented, runs on the AIX platform with an NFS file system installed.

```
#define _POSIX_SOURCE
#define INCL_32

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sqlenv.h>
#include <errno.h>
#include <sys/access.h>
#include <sys/flock.h>
#include <unistd.h>

#define BUF_SIZE 1500000 /* Local buffer to store the fetched records */
#define MAX_RECORD_SIZE 80 /* >= size of one written record */

int main(int argc, char *argv[]) {

 EXEC SQL INCLUDE SQLCA;
 EXEC SQL BEGIN DECLARE SECTION;
 char dbname[10]; /* Database name (argument of the program) */
 char userid[9];
 char passwd[19];
 char first_name[21];
 char last_name[21];
 char job_code[11];
 EXEC SQL END DECLARE SECTION;

 struct flock unlock ; /* structures and variables for handling */
 struct flock lock ; /* the NFS locking mechanism */
 int lock_command ;
 int lock_rc ;
 int iFileHandle ; /* output file */
 int iOpenOptions = 0 ;
 int iPermissions ;
 char * file_buf ; /* pointer to the buffer where the fetched
 records are accumulated */
 char * write_ptr ; /* position where the next record is written */
 int buffer_len = 0 ; /* length of used portion of the buffer */

 /* Initialization */

 lock.l_type = F_WRLCK; /* An exclusive write lock request */
 lock.l_start = 0; /* To lock the entire file */
 lock.l_whence = SEEK_SET;
 lock.l_len = 0;
 unlock.l_type = F_UNLCK; /* An release lock request */
 unlock.l_start = 0; /* To unlock the entire file */
 unlock.l_whence = SEEK_SET;
```

```

unlock.l_len = 0;
lock_command = F_SETLKW; /* Set the lock */
iOpenOptions = O_CREAT; /* Create the file if not exist */
iOpenOptions |= O_WRONLY; /* Open for writing only */

/* Connect to the database */

if (argc == 3) {
 /* get database name from the argument */
 strcpy(dbname, argv[2]);
 EXEC SQL CONNECT TO :dbname IN SHARE MODE ;
 if (SQLCODE != 0) {
 printf("CONNECT TO the database failed. SQLCODE = %ld",
 SQLCODE);
 exit(1);
 }
}
else if (argc == 5) {
 /* get database name from the argument */
 strcpy (dbname, argv[2]);
 strcpy (userid, argv[3]);
 strcpy (passwd, argv[4]);
 EXEC SQL CONNECT TO :dbname IN SHARE MODE USER :userid USING :passwd;
 if (SQLCODE != 0) {
 printf("Error: CONNECT TO the database failed. SQLCODE = %ld\n",
 SQLCODE);
 exit(1);
 }
}
else {
 printf ("\nUSAGE: largevol txt_file database [userid passwd]\n\n");
 exit(1);
} /* endif */

/* Open the input file with the specified access permissions */

if ((iFileHandle = open(argv[1], iOpenOptions, 0666)) == -1) {
 printf("Error: Could not open %s.\n", argv[2]);
 exit(2);
}

/* Set up error and end of table escapes */

EXEC SQL WHENEVER SQLERROR GO TO ext ;
EXEC SQL WHENEVER NOT FOUND GO TO cls ;

/* Declare and open the cursor */

EXEC SQL DECLARE c1 CURSOR FOR
 SELECT firstnme, lastname, job FROM employee
 WHERE workdept IS NOT NULL
 AND NODENUMBER(lastname) = CURRENT NODE;
EXEC SQL OPEN c1 ;

/* Set up the temporary buffer for storing the fetched result */

```

```

if ((file_buf = (char *) malloc(BUF_SIZE)) == NULL) {
 printf("Error: Allocation of buffer failed.\n");
 exit(3);
}
memset(file_buf, 0, BUF_SIZE); /* reset the buffer */
buffer_len = 0 ; /* reset the buffer length */
write_ptr = file_buf ; /* reset the write pointer */
/* For each fetched record perform the following */
/* - insert it into the buffer following the */
/* previously stored record */
/* - check if there is still enough space in the */
/* buffer for the next record and lock/write/ */
/* unlock the file and initialize the buffer */
/* if not */

do {
 EXEC SQL FETCH c1 INTO :first_name, :last_name, :job_code;
 buffer_len += sprintf(write_ptr, "%s %s %s\n",
 first_name, last_name, job_code);
 buffer_len = strlen(file_buf);
 /* Write the content of the buffer to the file if */
 /* the buffer reaches the limit */
 if (buffer_len >= (BUF_SIZE - MAX_RECORD_SIZE)) {
 /* get excl. write lock */
 lock_rc = fcntl(iFileHandle, lock_command, &lock);
 if (lock_rc != 0) goto file_lock_err;
 /* position at the end of file */
 lock_rc = lseek(iFileHandle, 0, SEEK_END);
 if (lock_rc < 0) goto file_seek_err;
 /* write the buffer */
 lock_rc = write(iFileHandle,
 (void *) file_buf, buffer_len);
 if (lock_rc < 0) goto file_write_err;
 /* release the lock */
 lock_rc = fcntl(iFileHandle, lock_command, &unlock);
 if (lock_rc != 0) goto file_unlock_err;
 file_buf[0] = '\0' ; /* reset the buffer */
 buffer_len = 0 ; /* reset the buffer length */
 write_ptr = file_buf ; /* reset the write pointer */
 }
 else {
 write_ptr = file_buf + buffer_len ; /* next write position */
 }
} while (1) ;

cls:
/* Write the last piece of data out to the file */
if (buffer_len > 0) {
 lock_rc = fcntl(iFileHandle, lock_command, &lock);
 if (lock_rc != 0) goto file_lock_err;
 lock_rc = lseek(iFileHandle, 0, SEEK_END);
 if (lock_rc < 0) goto file_seek_err;
 lock_rc = write(iFileHandle, (void *)file_buf, buffer_len);
 if (lock_rc < 0) goto file_write_err;
}

```

```

 lock_rc = fcntl(iFileHandle, lock_command, &unlock);
 if (lock_rc != 0) goto file_unlock_err;
 }
 free(file_buf);
close(iFileHandle);
EXEC SQL CLOSE c1;
exit (0);
ext:
 if (SQLCODE != 0)
 printf("Error: SQLCODE = %ld.\n", SQLCODE);
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL CONNECT RESET;
 if (SQLCODE != 0) {
 printf("CONNECT RESET Error: SQLCODE = %ld\n", SQLCODE);
 exit(4);
 }
 exit (5);
file_lock_err:
 printf("Error: file lock error = %ld.\n",lock_rc);
 /* unconditional unlock of the file */
 fcntl(iFileHandle, lock_command, &unlock);
 exit(6);
file_seek_err:
 printf("Error: file seek error = %ld.\n",lock_rc);
 fcntl(iFileHandle, lock_command, &unlock);
 /* unconditional unlock of the file */
 exit(7);
file_write_err:
 printf("Error: file write error = %ld.\n",lock_rc);
 fcntl(iFileHandle, lock_command, &unlock);
 /* unconditional unlock of the file */
 exit(8);
file_unlock_err:
 printf("Error: file unlock error = %ld.\n",lock_rc);
 fcntl(iFileHandle, lock_command, &unlock);
 /* unconditional unlock of the file */
 exit(9);
}

```

This method is applicable not only to a select from a single table, but also for more complex queries. If, however, the query requires noncollocated operations (that is, the Explain shows more than one subsection besides the Coordinator subsection), this can result in too many processes on some partitions if the query is run in parallel on all partitions. In this situation, you can store the query result in a temporary table TEMP on as many partitions as required, then do the final extract in parallel from TEMP.

If you want to extract all employees, but only for selected job classifications, you can define the TEMP table with the column names, FIRSTNAME, LASTNAME, and JOB, as follows:

```

INSERT INTO TEMP
SELECT FIRSTNAME, LASTNAME, JOB FROM EMPLOYEE WHERE WORKDEPT IS NOT NULL
AND EMPNO NOT IN (SELECT EMPNO FROM EMP_ACT WHERE
EMPNO<200)

```

Then you would do the parallel extract on TEMP.

When defining the TEMP table, consider the following:

- If the query specifies an aggregation GROUP BY, you should define the partitioning key of TEMP as a subset of the GROUP BY columns.
- The partitioning key of the TEMP table should have enough cardinality (that is, number of distinct values in the answer set) to ensure that the table is equally distributed across the partitions on which it is defined.
- Create the TEMP table with the NOT LOGGED INITIALLY attribute, then COMMIT the unit of work that created the table to release any acquired catalog locks.
- When you use the TEMP table, you should issue the following statements in a single unit of work:
  1. ALTER TABLE TEMP ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE (to empty the TEMP table and turn logging off)
  2. INSERT INTO TEMP SELECT FIRSTNAME...
  3. COMMIT

This technique allows you to insert a large answer set into a table without logging and without catalog contention. Note that any error in the unit of work that activated the NOT LOGGED state results in an unusable TEMP table. If this occurs, you will have to drop and recreate the TEMP table. For this reason, you should **not** use this technique to add data to a table that you could not easily recreate.

If you require the final answer set (which is the merged partial answer set from all nodes) to be sorted, you can:

- Specify the SORT BY clause on the final SELECT
- Do an extract into a separate file on each partition
- Merge the separate files into one output set using, for example, the sort -m AIX command

---

## Creating a Test Environment

You can create a test environment for your partitioned environment applications with DB2 Enterprise Edition, because DB2 Enterprise Edition enforces partitioning key restrictions like DB2 Enterprise - Extended Edition.

1. Create a model of your database design with DB2 Enterprise Edition.

2. Create sample tables with the PARTITIONING KEY clause that you will use to distribute your data across partitions in the production environment.
3. Develop and run your applications against the test database.

DB2 Enterprise Edition enforces the partitioning key constraints consistent with DB2 Enterprise - Extended Edition and provides a useful test environment for your applications.

---

## Error-Handling Considerations

In a partitioned environment, DB2 breaks up SQL statements into subsections, each of which are processed on the partition that contains the relevant data. As a result, an error may occur on a partition that does not have access to the application. This does not occur in a single-partition environment.

You should consider the following:

- Non-CURSOR (EXECUTE) non-severe errors
- CURSOR non-severe errors
- Severe errors
- Merged multiple SQLCA structures
- How to identify the partition that returned the error

If an application ends abnormally because of a severe error, indoubt transactions may be left in the database. (An indoubt transaction pertains to global transactions when one phase completes successfully, but the system fails before the a subsequent can complete, leaving the database in an inconsistent state.) For information on handling them, see the *Administration Guide*.

### Severe Errors

If a severe error occurs in DB2 Universal Database, one of the following will occur:

- The database manager on the node where the error occurs shuts down.  
Active units of work are not rolled back.  
In this situation, you must recover the node and any databases that were active on the node when the shutdown occurred.
- All agents are forced off the database at the node where the error occurred.  
All units of work on that database are rolled back.

In this situation, the database at the node where the error occurred is marked as inconsistent. Any attempt to access it results in either SQLCODE -1034 (SQLSTATE 58031) or SQLCODE -1015 (SQLSTATE 55025) being

returned. Before you or any other application on another node can access the database at this node, you must run the `RESTART DATABASE` command against the database. Refer to the *Command Reference* for information on this command.

The severe error `SQLCODE -1224 (SQLSTATE 55032)` can occur for a variety of reasons. If you receive this message, check the SQLCA, which will indicate which node failed. Then check the `db2diag.log` file shared between the nodes for details. See “Identifying the Partition that Returned the Error” on page 469 for additional information.

## Merged Multiple SQLCA Structures

One SQL statement may be executed by a number of agents on different nodes, and each agent may return a different SQLCA for different errors or warnings. The coordinating agent also has its own SQLCA. In addition, the SQLCA also has fields that indicate global numbers (such as the *sqlerrd* fields that indicate row counts). To provide a consistent view for applications, all the SQLCA values are merged into one structure. This structure is described in *SQL Reference*.

Error reporting is as follows:

- Severe error conditions are always reported. As soon as a severe error is reported, no additions beyond the severe error are added to the SQLCA.
- If no severe error occurs, a deadlock error takes precedence over other errors.
- For all other errors, the SQLCA for the first negative `SQLCODE` is returned to the application.
- If no negative `SQLCODEs` are detected, the SQLCA for the first warning (that is, positive `SQLCODE`) is returned to the application. The exception to this occurs if a data manipulation operation is issued on a table that is empty on one partition, but has data on other partitions. The `SQLCODE +100` is only returned to the application if agents from all partitions return `SQL0100W`, either because the table is empty on all partitions or there are no rows that satisfy the `WHERE` clause in an `UPDATE` statement.
- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- The values in the *sqlerrd* fields that indicate row counts are accumulations from all agents.

An application may receive a subsequent error or warning after the problem that caused the first error or warning is corrected. Errors are reported to the SQLCA to ensure that the first error detected is given priority over others. This ensures that an error caused by an earlier error cannot overwrite the



original error. Severe errors and deadlock errors are given higher priority because they require immediate action by the coordinating agent.

## Identifying the Partition that Returned the Error

If a partition returns an error or warning, its number is in the SQLERRD(6) field of the SQLCA. The number in this field is the same as that specified for the partition in the `db2nodes.cfg` file.

If an SQL statement or API call is successful, the partition number in this field is not significant.

For information about the SQLCA, see the *SQL Reference*.

---

## Debugging

You can use the tools described in the following sections for use in debugging your applications. For more information, refer to the *Troubleshooting Guide*.

### Diagnosing a Looping or Suspended application

It is possible that, after you start a query or application, you suspect that it is suspended (it does not show any activity) or that it is looping (it shows activity, but no results are returned to the application). Ensure that you have turned lock timeouts on. In some situations, however, no error is returned. In these situations, you may find the tools described in the *Troubleshooting Guide*, as well as the Database system monitor snapshot helpful.

One of the functions of the database system monitor that is useful for debugging applications is to display the status of all active agents. To obtain the greatest use from a snapshot, ensure that statement collection is being done before you run the application (preferably immediately after you run DB2START) as follows:

```
db2_a11 "db2 UPDATE MONITOR SWITCHES USING STATEMENT ON"
```

When you suspect that your application or query is either stalled or looping, issue the following command:

```
db2_a11 "db2 GET SNAPSHOT FOR AGENTS ON database"
```

Refer to the *System Monitor Guide and Reference* for information on how read the information collected from the snapshot, and for the details of using the database system monitor.



---

## Chapter 15. Writing Programs for DB2 Federated Systems

|                                          |     |                                          |     |
|------------------------------------------|-----|------------------------------------------|-----|
| Introduction to DB2 Federated Systems    | 471 | A Request with a Subquery . . . .        | 480 |
| Accessing Data Source Tables and Views   | 472 | A Request with Set Operators . . . .     | 480 |
| Working with Nicknames . . . . .         | 472 | A Request for a Join. . . . .            | 481 |
| Cataloging Information about Data        |     | Using Server Options to Facilitate       |     |
| Source Tables and Views . . . . .        | 472 | Optimization . . . . .                   | 481 |
| Considerations and Restrictions . . . .  | 473 | Invoking Data Source Functions. . . . .  | 482 |
| Defining Column Options. . . . .         | 474 | Enabling DB2 to Invoke Data Source       |     |
| Using Nicknames with Views . . . . .     | 475 | Functions . . . . .                      | 482 |
| Using Isolation Levels to Maintain Data  |     | Reducing the Overhead of Invoking a      |     |
| Integrity. . . . .                       | 476 | Function . . . . .                       | 483 |
| Working with Data Type Mappings . . . .  | 477 | Specifying Function Names in the         |     |
| How DB2 Determines What Data Types       |     | CREATE FUNCTION MAPPING                  |     |
| to Define Locally. . . . .               | 477 | Statement . . . . .                      | 484 |
| Default Data Type Mappings. . . . .      | 477 | Discontinuing Function Mappings . . . .  | 484 |
| How You Can Override Default Type        |     | Using Pass-Through to Query Data Sources |     |
| Mappings and Create New Ones . . . .     | 478 | Directly . . . . .                       | 485 |
| Defining a Type Mapping That             |     | SQL Processing in Pass-Through Sessions  | 485 |
| Applies to One or More Data Sources      | 478 | Considerations and Restrictions . . . .  | 486 |
| Changing a Type Mapping for a            |     | Using Pass-Through with All Data         |     |
| Specific Table . . . . .                 | 478 | Sources . . . . .                        | 486 |
| Using Distributed Requests to Query Data |     | Using Pass-Through with Oracle Data      |     |
| Sources . . . . .                        | 479 | Sources . . . . .                        | 486 |
| Coding Distributed Requests. . . . .     | 479 |                                          |     |

---

### Introduction to DB2 Federated Systems

A **DB2 federated system** is a distributed computing system that consists of:

- A DB2 server, called a **federated server**.
- Multiple semi-autonomous data sources that the federated server sends queries to. Each data source consists of an instance of a relational database management system, plus the database or databases that the instance supports. The data sources in a DB2 federated system can include Oracle instances and instances of the members of the DB2 family.

To client applications, the data sources appear as a single collective database. However, the applications actually interface with a database, called the **federated database**, that is within the federated server. To obtain data from data sources, they submit queries in DB2 SQL to the federated database. DB2 then distributes the queries to the appropriate data sources, collects the requested data, and returns this data to the applications.

Applications can use DB2 SQL to request values of any data types that DB2 can recognize, except for LOB data types. To write to a data source—for

example, to update a data source table—an application must use the data source’s own SQL in a special mode called pass-through.

The federated database’s system catalog contains information not only about the objects in the database, but also about the data sources and certain tables, views, and functions in them. The catalog, then, contains information about the entire federated system; accordingly, it is called a **global catalog**.

For a high-level overview of DB2 federated systems, see the *Administration Guide, Design and Implementation*. For an extended overview, see the *SQL Reference*. For examples of DB2 SQL queries that an application can submit, see “Using Distributed Requests to Query Data Sources” on page 479. For information about Pass-Through, see “Using Pass-Through to Query Data Sources Directly” on page 485.

---

## Accessing Data Source Tables and Views

This section provides information to help you access and use data source tables and views. The topics covered are:

- Nicknames that you assign to the tables and views, so that the federated server can reference them.
- Isolation levels that help you to maintain data integrity at the data source when you access the tables and views

### Working with Nicknames

A **nickname** is an identifier by which an application can reference a data source table or view. This section:

- Explains how the global catalog can be supplied with information about tables that you create nicknames for
- Lists considerations and restrictions to remember when you work with nicknames
- Describes parameters that you can set to optimize queries
- Discusses ways to use views referenced by nicknames

### Cataloging Information about Data Source Tables and Views

When a nickname is created for a data source table or view, DB2 updates the global catalog with information that the optimizer can use in planning how to retrieve data from the table or view. This information includes, for example, the name of the table or view and the names and attributes of the table’s or view’s columns.

In the case of a table, the information also includes:

- Statistics (for example, the number of rows and the number of pages on which the rows exist). To ensure that DB2 obtains the latest statistics, it is advisable to run the data source's equivalent of the RUNSTATS command against the table before you create the nickname.
- Descriptions of any indexes that the table has. If the table has no indexes, you can nonetheless supply the catalog with metadata that an index definition typically contains—for example, which column or columns in the table have unique values, and whether any rows are unique. You can generate this metadata, which is collectively called an **index specification**, by running the CREATE INDEX statement against the table's nickname. Be aware that in this case the statement produces only the index specification; it does not create an actual index. For documentation on this statement, see the *SQL Reference*.

To find out what information about a data source table is stored in the global catalog, query the SYSCAT.TABLES and SYSCAT.COLUMNS catalog view. To find out what information about a table's index is stored in the catalog, or what a particular index specification contains, query the SYSCAT.INDEXES catalog view. For descriptions of these views, see the *SQL Reference*. For further discussion about updating the global catalog with information about tables and indexes, see the *Administration Guide, Performance*.

### Considerations and Restrictions

There are several considerations and restrictions to bear in mind when you:

- Define, change, and drop nicknames
- Reference tables and views by their nicknames
- Perform operations on tables and views that are referenced by nicknames

### Defining, Changing, and Dropping Nicknames:

- To define a nickname for a table or view, use the CREATE NICKNAME statement. In this statement:
  - You reference an Oracle table or view by its name.
  - You can reference a DB2 family table or view by its name, or, if it has an alias, by this alias.
- You can define more than one nickname for the same table or view. You can also define an alias for a nickname with the CREATE ALIAS statement.
- To change a nickname, you must drop it and then replace it. To drop it, use the DROP NICKNAME statement; to replace it, use the CREATE NICKNAME statement.
- Dropping a nickname causes any views defined using the nickname to be inoperative and invalidates any plans that are dependent upon it.

For documentation on the CREATE NICKNAME, CREATE ALIAS, and DROP NICKNAME statements, see the *SQL Reference*.

#### Referencing Tables and Views by Nickname:

- After a data source table or view has been given a nickname, you can reference the table or view by that nickname only (except in a pass-through session). For example, if you define the nickname DEPT to represent a table called DB2MVS1.PERSON.DEPT, the statement SELECT \* FROM DEPT is allowed, but SELECT \* FROM DB2MVS1.PERSON.DEPT is not allowed. In a pass-through session, however, you must access a table or view by its data source name.
- You cannot reference a nickname in the CREATE TRIGGER statement.
- If you reference a nickname in the summary-table-definition clause of the CREATE TABLE statement, you must also specify the DEFINITION ONLY keywords in this clause.

#### Performing Operations on Tables and Views That Have Nicknames:

- The COMMENT ON statement is valid against a nickname and columns that are defined on a nickname. This statement updates the global catalog; it does not update data source catalogs.
- GRANT and REVOKE statements are valid against a nickname for certain privileges and for all users and groups. However, DB2 does not issue a corresponding GRANT or REVOKE against the table or view that the nickname references. For more information about nickname privileges, see the *Administration Guide, Design and Implementation*.
- Data sources are read-only. Therefore:
  - INSERT, UPDATE, and DELETE statements are not valid against nicknames.
  - A view that contains a UNION ALL clause for a nickname cannot be updated.
- You cannot run the DB2 utilities (RUNSTATS, IMPORT, EXPORT and so on) against nicknames.

#### Defining Column Options

When you define a nickname for a table or view, you can provide the global catalog with information about particular columns in the table or view. You specify this information in the form of values that you assign to parameters called **column options**. You can specify any of these values in either upper- or lowercase. Table 16 on page 475 describes the column options and their values.

Table 16. Column Options and Their Settings

| Option                     | Valid Settings                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                | Default Setting |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| numeric_string             | 'y'                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Yes, this column contains only strings of numeric data. IMPORTANT: If this column contains only numeric strings followed by trailing blanks, it is inadvisable to specify 'y'. | 'n'             |
|                            | 'n'                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | No, this column is not limited to strings of numeric data.                                                                                                                     |                 |
|                            | By setting numeric_string to 'y' for a column, you are informing the optimizer that this column contains no blanks that could interfere with sorting of the column's data.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                |                 |
|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                |                 |
| varchar_no_trailing_blanks | Indicates whether trailing blanks are absent from a specific VARCHAR column:                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                | 'n'             |
|                            | 'y'                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Yes, trailing blanks are absent from this VARCHAR column.                                                                                                                      |                 |
|                            | 'n'                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | No, trailing blanks are not absent from this VARCHAR column.                                                                                                                   |                 |
|                            | If data source VARCHAR columns contain no padded blanks, then the optimizer's strategy for accessing them depends in part on whether they contain trailing blanks. By default, the optimizer "assumes" that they actually do contain trailing blanks. On this assumption, it develops an access strategy that involves modifying queries so that the values returned from these columns are the ones that the user expects. If, however, a VARCHAR column has no trailing blanks, and you let the optimizer know this, it can develop a more efficient access strategy. To tell the optimizer that a specific column has no trailing blanks, specify that column in the ALTER NICKNAME statement (for guidelines, see the SQL Reference). |                                                                                                                                                                                |                 |

You set column options in the ALTER NICKNAME statement. For information about this statement, see the *SQL Reference*.

Using Nicknames with Views

You can use nicknames with views in two main ways:

- You can create nicknames for data source views. The federated server treats the nickname of a data source view the same way it treats the nickname of a data source table.
- You can create federated database views of data source tables and views that have nicknames. For example, because the federated server can accommodate a join of base tables at different locations, you can easily

define federated database views of base tables that reside at different data sources. Such multi-location views offer a high degree of data independence for a globally integrated database, just as views defined on multiple local tables do for centralized relational database managers. This global view mechanism is one way in which the federated server offers a high degree of data independence.

The action of creating a federated database view of data source data is sometimes called “creating a view on a nickname”. This phrase reflects the fact that for the view to be created, the CREATE VIEW statement’s fullselect must reference the nickname of each table and view that the view is to contain.

Views do not have statistics or indexes of their own because they are not actual tables located in a database. This statement is true even when a view is identical in structure and content to a single base table. For more information about statistics and indexes, see *Administration Guide, Performance*.

## Using Isolation Levels to Maintain Data Integrity

You can maintain data integrity for a data source table by requesting that the table’s rows be locked at a specific isolation level. For example, to ensure that you have sole access to a row, you would specify the repeatable read (RR) isolation level for that row.

The federated server maps the isolation level you request to a corresponding one at the data source. To illustrate this, Table 17 lists:

- The isolation levels that you can request. They are:
  - CS      Cursor stability
  - RR      Repeatable read
  - RS      Read stability
  - UR      Uncommitted read
- The Oracle isolation levels that the requested levels map to.

Table 17. Comparable Isolation Levels between the Federated Server and Oracle Data Sources.

| <b>Federated Server (DB2)</b> | CS      | RR                    | RS                    | UR                       |
|-------------------------------|---------|-----------------------|-----------------------|--------------------------|
| <b>Oracle</b>                 | Default | Transaction read-only | Transaction read-only | Same as cursor stability |



---

## Working with Data Type Mappings

When you create a nickname for a data source table, DB2 populates the global catalog with information about the table. This information includes, but is not limited to, the nickname, the table's name, all column names and, for each column:

- The data type that was defined for the column at the data source. (This section calls this type a **remote** type.)
- A corresponding data type that is supported by DB2 and registered to the federated database. (This section calls this type a **local** type.)

This section explains how DB2 uses data type mappings to determine what DB2-supported data type should be defined for the column of a data source table. The section then discusses data type mappings (sometimes called “type mappings”, for short) in two subsections. The first describes default mappings; the second shows how you can override default mappings and create new mappings.

### How DB2 Determines What Data Types to Define Locally

How does DB2 determine what local type to use for a remote column? It consults a mapping between the column's type at the data source and a comparable local type, and chooses the latter. For example, in a default mapping supplied by DB2, the DB2 for VSE & VM data type CHAR, which supports up to 254 bytes, points to the DB2 data type CHAR. So if you create a nickname for a DB2 for VSE & VM table, and column C1 of the table has a data type of CHAR with a maximum length of 200, then, unless you override the default, the DB2 type CHAR is defined locally for C1.

### Default Data Type Mappings

Because of differences between RDBMSs, a default mapping between a data source data type and a federated server data type is not always one-to-one. However, the mapping is close enough to ensure that all requested values are returned.

For example, there is a default type mapping between:

- The Oracle type NUMBER(9,0) (where 9 is the maximum precision and 0 the maximum scale)
- The DB2 type INTEGER, with a maximum length of 4 bytes

Suppose that you create a nickname for an Oracle table that has a column C2 with a type of NUMBER(9,0). If you do not change the default mapping, the type for C2 will be locally defined as INTEGER. And because the 4 bytes of

INTEGER support a maximum precision of 10, you can be sure that all values of C2 will be returned when C2 is queried from the federated server.

For listings of the default data type mappings, see the *SQL Reference*.

## How You Can Override Default Type Mappings and Create New Ones

As the preceding example indicates, the local type and remote type in a default mapping are similar enough to ensure that when you query remote columns for which the remote type is defined, all values that conform to both types will be returned. But sometimes, you might require an alternative mapping. Consider these scenarios:

### Defining a Type Mapping That Applies to One or More Data Sources

Certain columns in three tables in an Oracle data source have a data type DATE for time stamps. In a default mapping, this type points to the local DB2 type TIMESTAMP. So if you were to create nicknames for the three tables without changing the default, TIMESTAMP would be defined locally for these columns, and DB2 queries of the columns would yield time stamps. But suppose that you want such queries to yield times only. You could then map Oracle DATE to the DB2 type TIME, overriding the default. That way, when you create the nicknames, TIME, not TIMESTAMP, would be defined locally for the columns. As a result, when you query them, only the time portion of the time stamps would be returned. To override the default type mapping, you would use the CREATE TYPE MAPPING statement.

In the CREATE TYPE MAPPING statement, you can indicate whether the new mapping that you want is to apply to a specific data source (for example, a data source that a department in your organization uses) or to all data sources of a specific type (for example, all Oracle data sources), or to all data sources of a specific version of a type (for example, all Oracle 8.0.3 data sources).

### Changing a Type Mapping for a Specific Table

You can change the local type in a type mapping for a specific table. For example, Oracle data type NUMBER(32,3) maps by default to the DB2 data type DOUBLE, a floating decimal data type. Suppose that in an Oracle table for employee information, a column BONUS was defined with a data type of NUMBER(32,3). Because of the mapping, a query of BONUS could return values that look like this:

```
5.00000000000000E+002
1.00000000000000E+003
```

where +002 signifies that the decimal point should be moved two places to the right, and +003 signifies that the decimal point should be moved three places to the right.

So that queries of BONUS can return values that look like dollar amounts, you could, for this particular table, remap NUMBER(32,3) to a DB2 DECIMAL type with a precision and scale that reflect the format of actual bonuses. For example, if you knew that the dollar portion of the bonuses would not exceed six figures, you could remap NUMBER(32,3) to DECIMAL(8,2). Under the constraint of this new mapping, a query of BONUS would return values like this:

```
500.00
1000.00
```

To change the type mapping for a column of a specific table, use the ALTER NICKNAME statement. With this statement, you can change the type defined locally for a column of a table for which a nickname has been defined.

---

## Using Distributed Requests to Query Data Sources

Queries submitted to the federated database can request results that are yielded by a single data source; but typically they request results that are yielded by multiple data sources. Because a typical query is distributed to multiple data sources, it is called a **distributed request**.

This section:

- Illustrates ways to code distributed requests
- Introduces you to a way to abet optimization of certain distributed requests

### Coding Distributed Requests

In general, a distributed request uses one or more of three SQL conventions to specify where data is to be retrieved from: subqueries, set operators, and join subselects. This section provides examples within the context of the following scenario: A federated server is configured to access a DB2 Universal Database for OS/390 data source, a DB2 Universal Database for AS/400 data source, and an Oracle data source. Stored in each data source is a table that contains employee information. The federated server references these tables by nicknames that point to where the tables reside: UDB390\_EMPLOYEES, AS400\_EMPLOYEES, and ORA\_EMPLOYEES. (Nicknames do not have to reference data sources; the ones in this scenario do so only to underline the point that the tables reside in different RDBMSs.) In addition to ORA\_EMPLOYEES, the Oracle data source has a table, nicknamed ORA\_COUNTRIES, that contains information about the countries that the employees live in.

## A Request with a Subquery

Table AS400\_EMPLOYEES contains the phone numbers of employees who live in Asia. It also contains the country codes associated with these phone numbers, but it does not list the countries that the codes represent. Table ORA\_COUNTRIES, however, does list both codes and countries. The following query uses a subquery to find out the country code for China; and it uses SELECT and WHERE clauses to list those employees in AS400\_EMPLOYEES whose phone numbers require this particular code.

```
SELECT NAME, TELEPHONE
FROM DJADMIN.AS400_EMPLOYEES
WHERE COUNTRY_CODE IN
(SELECT COUNTRY_CODE
FROM DJADMIN.ORA_COUNTRIES
WHERE COUNTRY_NAME = 'CHINA')
```

## A Request with Set Operators

The federated server supports three set operators:

- UNION  
Use this set operator to combine the rows that satisfy any of two or more SELECT statements.
- EXCEPT  
Use this set operator to retrieve those rows that satisfy the first SELECT statement but not the second.
- INTERSECT  
Use this set operator to retrieve those rows that satisfy both SELECT statements.

All three set operators might have the ALL operand to indicate that duplicate rows are not be removed from the result, thus eliminating the need for an extra sort.

The following query retrieves all employee names and country codes that are present in both the AS400\_EMPLOYEES and UDB390\_EMPLOYEES tables, even though each table resides in a different data source.

```
SELECT NAME, COUNTRY_CODE
FROM AS400_EMPLOYEES
INTERSECT
SELECT NAME, COUNTRY_CODE
FROM UDB390_EMPLOYEES
```

## A Request for a Join

A relational join produces a result set that contains a combination of columns retrieved from two or more tables. Be aware that you should specify conditions to limit the size of the result set's rows.

The query below combines employee names and their corresponding country names by comparing the country codes listed in two tables. Each table resides in a different data source.

```
SELECT T1.NAME, T2.COUNTRY_NAME
FROM DJADMIN.AS400_EMPLOYEES T1, DJADMIN.ORA_COUNTRIES T2
WHERE T1.COUNTRY_CODE = T2.COUNTRY_CODE
```

## Using Server Options to Facilitate Optimization

Federated system users can use parameters called **server options** to supply the global catalog with information that applies to a data source as a whole, or to control interaction between DB2 and a data source. For example, to catalog the identifier of the instance that serves as the basis of a data source, the database administrator assigns that identifier as a value to the server option “node”.

Several server options address a major area of interaction between DB2 and data sources: optimization of queries. For example, just as you can use the column option “varchar\_no\_trailing\_blanks” to inform the DB2 optimizer of specific data source VARCHAR columns that have no trailing blanks, so can you use a server option—also called “varchar\_no\_trailing\_blanks”—to inform the optimizer of data sources whose VARCHAR columns are all free of trailing blanks. For a summary of how such information helps the optimizer to create an access strategy, see Table 16 on page 475.

In addition, you can set the server option “plan\_hints” to a value that enables DB2 to provide Oracle data sources with statement fragments, called **plan hints**, that help Oracle optimizers do their job. Specifically, plan hints can help an optimizer to decide matters such as which index to use in accessing a table, and which table join sequence to use in retrieving data for a result set.

Typically, the database administrator sets server options for a federated system. However, a programmer can make good use of those options that help to optimize queries. For example, suppose that for data sources ORACLE1 and ORACLE2, the plan\_hints server option is set to its default, ‘n’ (no, do not furnish this data source with plan hints). Also suppose that you write a distributed request for data from ORACLE1 and ORACLE2, and that you expect that plan hints would help the optimizers at these data sources improve their strategies for accessing this data. You could override the default with a setting of ‘y’ (yes, furnish the plan hints) while your application is

connected to the federated database. When the connection is completed, the setting would automatically revert to 'n'.

To enforce a server option setting for the duration of a connection to the federated database, use the SET SERVER OPTION statement. To ensure that the setting takes effect, you must specify the statement right after the CONNECT statement. In addition, it is advisable to prepare the statement dynamically.

For documentation of the SET SERVER OPTION statement, see the *SQL Reference*. For descriptions of all server options and their settings, see the *Administration Guide, Performance*.

---

## Invoking Data Source Functions

This section explains how you can:

- Enable DB2 to invoke a data source function that it does not recognize
- Help to reduce the overhead consumed when DB2 invokes a function
- Specify names of functions that you want to map to one another
- Discontinue the use of a mapping between functions

### Enabling DB2 to Invoke Data Source Functions

At times you might want DB2 to invoke a data source function that it does not recognize. Such a function might be a user-defined function or a new built-in function that is unknown to DB2.

Before DB2 can access a data source function that it does not recognize, you must create a mapping between this function and a counterpart that is stored in the federated database. To create the mapping, select the counterpart and submit the DDL statement for creating the mapping. This statement is called CREATE FUNCTION MAPPING.

The counterpart can be an existing function or function template, or a function or function template that you create. (A **function template** is a partial function that has no executable code.) You can create a function or function template with the CREATE FUNCTION statement.

The data source function and its federated database counterpart should correspond in the following ways:

- Both should have the same number of input parameters.
- The data types of the input parameters of the data source function should be compatible with the data types of the input parameters of the federated database counterpart.

For documentation on the CREATE FUNCTION MAPPING and CREATE FUNCTION statements, see the *SQL Reference*.

## Reducing the Overhead of Invoking a Function

The DDL for mapping a federated server function to a data source function—the CREATE FUNCTION MAPPING statement—can include estimated statistics on the overhead that would be consumed when the data source function is invoked. For example, the statement can specify the estimated number of instructions that would be required to invoke the data source function, and the estimated number of I/Os that would be expended for each byte of the argument set that is passed to this function. These estimates are stored in the global catalog; you can see them in the SYSCAT.FUNCMAPOPTIONS view. In addition, if a DB2 function (rather than a function template) participates in the mapping, the catalog contains estimates of overhead that would be consumed when this function is invoked. You can see these latter estimates in the SYSCAT.FUNCTIONS view.

After the mapping is created, you can submit distributed requests that reference the DB2 function. For example, if you mapped a DB2 user-defined function called DOLLAR to an Oracle user-defined function called US\_DOLLAR, your request would specify DOLLAR rather than US\_DOLLAR. When the request is processed, the optimizer evaluates multiple access strategies. Some of them reflect the estimated overhead of invoking the DB2 function; others reflect the estimated overhead of invoking the data source function. The strategy that is expected to cost the least amount of overhead is the one that is used.

If any estimates of consumed overhead change, you can record the change in the global catalog. To record new estimates for the data source function, first drop or disable the function mapping (for information about how to do this, see “Discontinuing Function Mappings” on page 484). Then recreate the mapping with the CREATE FUNCTION MAPPING statement, specifying the new estimates in the statement. When you run the statement, the new estimates will be added to the SYSCAT.FUNCTIONS catalog view. To record changed estimates for the DB2 function, update the SYSSTAT.FUNCTIONS catalog view directly.

You specify estimated statistics in the CREATE FUNCTION MAPPING statement by assigning them as values to parameters called **function mapping options**. Table 18 on page 484 describes these options and their values.

Table 18. Function Mapping Options and Their Settings

| Option            | Valid Settings                                                                                                           | Default Setting |
|-------------------|--------------------------------------------------------------------------------------------------------------------------|-----------------|
| ios_per_invoc     | Estimated number of I/Os per invocation of a data source function.                                                       | '0'             |
| insts_per_invoc   | Estimated number of instructions processed per invocation of the data source function.                                   | '450'           |
| ios_per_argbyte   | Estimated number of I/Os expended for each byte of the argument set that is passed to the data source function.          | '0'             |
| insts_per_argbyte | Estimated number of instructions processed for each byte of the argument set that is passed to the data source function. | '0'             |
| percent_argbytes  | Estimated average percent of input argument bytes that the data source function will actually read.                      | '100'           |
| initial_ios       | Estimated number of I/Os performed the first and last time that the data source function is invoked.                     | '0'             |
| initial_insts     | Estimated number of instructions processed the first and last time that the data source function is invoked.             | '0'             |

For more information about the DROP FUNCTION MAPPING statement, the SYSCAT.FUNCTIONS and SYSSTAT.FUNCTIONS views, and the SYSCAT.FUNCMAPOPTIONS view, see the *SQL Reference*.

## Specifying Function Names in the CREATE FUNCTION MAPPING Statement

How you code the CREATE FUNCTION MAPPING statement depends on part on whether the names of the objects that you are mapping together are the same or different. If you are creating a mapping between two functions (or a function template and a function) that have the same name, you must assign this name to the *function-name* parameter.

But if the names differ, then:

- Assign the name of the federated database function or function template to the *function-name* parameter.
- Specify a function mapping option called “remote\_name” and assign the name of the data source function to this option. The name must have fewer than 255 characters.

## Discontinuing Function Mappings

If you want to discontinue using a function mapping, follow these guidelines:



- If the mapping is listed in the SYSCAT.FUNCMAPPINGS catalog view, delete the mapping. You do this with the DROP FUNCTION MAPPING statement.
- To discontinue a default mapping that is not listed in the SYSCAT.FUNCMAPPINGS view, you disable the mapping. You do this in the CREATE FUNCTION MAPPING statement by setting a function mapping option called “disable” to ‘y’ (yes, disable this function mapping). The default is ‘n’.

---

## Using Pass-Through to Query Data Sources Directly

You can use a facility called **pass-through** to query a data source in the SQL that is native to that data source. This section:

- States what kind of SQL statements a federated server and its associated data sources process in pass-through sessions.
- Lists considerations and restrictions to be aware of when you use pass-through.

### SQL Processing in Pass-Through Sessions

The following rules specify whether an SQL statement is processed by DB2 or by a data source:

- DB2 processes all static SQL statements that are submitted in a pass-through session.
- The data source processes dynamic SQL statements that undergo a PREPARE within a pass-through session.
- DB2 processes dynamic SQL statements that do not undergo a PREPARE within a pass-through session.
- If you issue the COMMIT or ROLLBACK command during a pass-through session, this command will complete the current unit of work (UOW). For example, in the following UOW, a nickname is created and then a pass-through session is opened to retrieve a table’s data. A COMMIT is entered during this session, but it applies to the entire UOW. (In this example, the name of the federated database is DJDB and the name of the table for which a nickname is being created is T1.)

```
CONNECT TO DJDB
CREATE NICKNAME NICKY FOR T1
SET PASSTHRU DJDB
SELECT * FROM T2
COMMIT
SET PASSTHRU RESET
```

## Considerations and Restrictions

There are a number of considerations and restrictions to bear in mind when you use pass-through. Some of them are of a general nature; others apply to Oracle data sources only.

### Using Pass-Through with All Data Sources

The following information applies to all data sources:

- You can use pass-through to write to data sources; for example, to insert, update, and delete table rows. But note that you cannot use WHERE CURRENT OF conditions in UPDATE and DELETE statements within a pass-through session.
- An application can have several SET PASSTHRU statements in effect at the same time to different data sources. Although the application might have issued multiple SET PASSTHRU statements, the pass-through sessions are not truly nested. The federated server will not pass through one data source to access another. Rather, the server accesses each data source directly.
- If multiple pass-through sessions are open at the same time, be sure to issue a COMMIT each time you want to conclude a unit of work in each session. Then, when you need to terminate the sessions, you can do so with a single SET PASSTHRU RESET statement.
- Host variables defined in SQL statements within a pass-through session must take the form :Hn where H is uppercase and n is a unique whole number. The values of n must be numbered consecutively beginning with zero.
- You cannot pass through to more than one data source at a time.
- Pass-through does not support stored procedure calls.

### Using Pass-Through with Oracle Data Sources

The following information applies to Oracle data sources:

- The following restriction applies when a remote client issues a SELECT statement from a command line processor (CLP) in pass-through mode: If the client code is a DB2 SDK prior to DB2 Universal Database Version 5, the SELECT will elicit an SQLCODE -30090 with reason code 11. To avoid this error, remote clients must use a DB2 SDK that is at Version 5 or greater.
- Any DDL statement issued against an Oracle server is performed at parse time and is not subject to transaction semantics. The operation, when complete, is automatically committed by Oracle. If a rollback occurs, the DDL is not rolled back.

- When you issue a `SELECT` statement from raw data types, use the `RAWTOHEX` function to receive the hexadecimal values. When you perform an `INSERT` into raw data types, provide the hexadecimal representation.



---

## Part 6. Language Considerations



---

# Chapter 16. Programming in C and C++

|                                             |     |                                           |     |
|---------------------------------------------|-----|-------------------------------------------|-----|
| Programming Considerations . . . . .        | 491 | Host Structure Support. . . . .           | 511 |
| Language Restrictions . . . . .             | 491 | Indicator Tables . . . . .                | 513 |
| Trigraph Sequences . . . . .                | 491 | Null-terminated Strings . . . . .         | 514 |
| C++ Type Decoration Consideration           | 492 | Pointer Data Types . . . . .              | 516 |
| Input and Output Files. . . . .             | 492 | Using Class Data Members as Host          |     |
| Include Files . . . . .                     | 493 | Variables . . . . .                       | 517 |
| Including Files . . . . .                   | 496 | Using Qualification and Member            |     |
| Embedding SQL Statements . . . . .          | 497 | Operators . . . . .                       | 518 |
| Host Variables. . . . .                     | 498 | Handling Graphic Host Variables . . . . . | 519 |
| Naming Host Variables. . . . .              | 498 | Multi-byte Character Encoding . . . . .   | 519 |
| Declaring Host Variables . . . . .          | 499 | Selecting the wchar_t or sqldbchar        |     |
| Indicator Variables . . . . .               | 503 | Data Type . . . . .                       | 520 |
| Graphic Host Variable Declarations in C     |     | The WCHARTYPE Precompiler                 |     |
| or C++ . . . . .                            | 503 | Option . . . . .                          | 521 |
| LOB Data Declarations in C or C++ . . . . . | 506 | Japanese or Traditional Chinese EUC,      |     |
| LOB Locator Declarations in C or C++        | 508 | and UCS-2 Considerations . . . . .        | 524 |
| File Reference Declarations in C or C++     | 509 | Supported SQL Data Types . . . . .        | 525 |
| Initializing Host Variables. . . . .        | 510 | FOR BIT DATA . . . . .                    | 530 |
| C Macro Expansion . . . . .                 | 510 | SQLSTATE and SQLCODE Variables . . . . .  | 530 |

---

## Programming Considerations

Special host language programming considerations are discussed in the following sections. Included is information on language restrictions, host-language-specific include files, embedding SQL statements, host variables, and supported data types for host variables.

---

## Language Restrictions

The following sections describe the C/C++ language restrictions.

### Trigraph Sequences

Some characters from the C or C++ character set are not available on all keyboards. These characters can be entered into a C or C++ source program using a sequence of three characters called a *trigraph*. Trigraphs are not recognized in SQL statements. The precompiler recognizes the following trigraphs within host variable declarations:

| Trigraph | Definition        |
|----------|-------------------|
| ??(      | Left bracket '['  |
| ??)      | Right bracket ']' |

|     |                 |
|-----|-----------------|
| ??< | Left brace '{'  |
| ??> | Right brace '}' |

The remaining trigraphs listed below may occur elsewhere in a C or C++ source program:

| Trigraph | Definition       |
|----------|------------------|
| ??=      | Hash mark '#'    |
| ??/      | Back slash '\'   |
| ??'      | Caret '^'        |
| ??!      | Vertical Bar ' ' |
| ??-      | Tilde '~'        |

## C++ Type Decoration Consideration

When writing a stored procedure or a UDF using C++, you may want to consider declaring the procedure or UDF as:

```
extern "C" ...procedure or function declaration...
```

The extern "C" prevents type decoration of the function name by the C++ compiler. Without this declaration, you have to include all the type decoration for the function name when you call the stored procedure, or issue the CREATE FUNCTION statement.

---

## Input and Output Files

By default, the input file can have the following extensions:

- .sqc** For C files on all supported platforms
- .sqC** For C++ files on UNIX platforms
- .sqx** For C++ files on OS/2 and Windows 32-bit operating systems

By default, the corresponding precompiler output files have the following extensions:

- .c** For C files on all supported platforms
- .C** For C++ files on UNIX platforms
- .cxx** For C++ files on OS/2 and Windows 32-bit operating systems



You can use the OUTPUT precompile option to override the name and path of the output modified source file. If you use the TARGET C or TARGET CPLUSPLUS precompile option, the input file does not need a particular extension.

---

## Include Files

The host-language-specific include files (header files) for C and C++ have the file extension `.h`. The include files that are intended to be used in your applications, are described below.

### **SQL (`sql.h`)**

This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

### **SQLADEF (`sqladef.h`)**

This file contains function prototypes used by precompiled C and C++ applications.

### **SQLAPREP (`sqlaprep.h`)**

This file contains definitions required to write your own precompiler.

### **SQLCA (`sqlca.h`)**

This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

### **SQLCLI (`sqlcli.h`)**

This file contains the function prototypes and constants needed to write a Call Level Interface (DB2 CLI) application. The functions in this file are common to both X/Open Call Level Interface and ODBC Core Level.

### **SQLCLI1 (`sqlcli1.h`)**

This file contains the function prototypes and constants needed to write a Call Level Interface (DB2 CLI) that makes use of the more advanced features in DB2 CLI. Many of the functions in this file are common to both X/Open Call Level Interface and ODBC Level 1. In addition, this file also includes X/Open-only functions and DB2-specific functions.

This file includes both `sqlcli.h` and `sqlext.h` (which contains ODBC Level2 API definitions).

### **SQLCODES (`sqlcodes.h`)**

This file defines constants for the SQLCODE field of the SQLCA structure.

**SQLDA (sqlda.h)**

This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

**SQLEAU (sqleau.h)**

This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

**SQLENV (sqlenv.h)**

This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

**SQLTEXT (sqltext.h)**

This file contains the function prototypes and constants of those ODBC Level 1 and Level 2 APIs that are not part of the X/Open Call Level Interface specification and is therefore used with the permission of Microsoft Corporation.

**SQL819A (sqle819a.h)**

If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL819B (sqle819b.h)**

If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQL850A (sqle850a.h)**

If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL850B (sqle850b.h)**

If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQL932A (sqle932a.h)**

If the code page of the database is 932 (ASCII Japanese), this sequence

sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL932B (sqle932b.h)**

If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQLJACB (sqljacob.h)**

This file defines constants, structures and control blocks for the DB2 Connect interface.

**SQLMON (sqlmon.h)**

This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

**SQLSTATE (sqlstate.h)**

This file defines constants for the SQLSTATE field of the SQLCA structure.

**SQLSYSTEM (sqlsystem.h)**

This file contains the platform-specific definitions used by the database manager APIs and data structures.

**SQLUDF (sqludf.h)**

This file defines constants and interface structures for writing User Defined Functions (UDFs). For more information on this file, see “The UDF Include File: sqludf.h” on page 311.

**SQLUTIL (sqlutil.h)**

This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

**SQLUV (sqluv.h)**

This file defines structures, constants, and prototypes for the asynchronous Read Log API, and APIs used by the table load and unload vendors.

**SQLUVEND (sqluvend.h)**

This file defines structures, constants and prototypes for the APIs to be used by the storage management vendors.

**SQLXA (sqlxa.h)**

This file contains function prototypes and constants used by applications that use the X/Open XA Interface.

## Including Files

There are two methods for including files: the EXEC SQL INCLUDE statement and the #include macro. The precompiler will ignore the #include, and only process files included with the EXEC SQL INCLUDE statement.

To locate files included using EXEC SQL INCLUDE, the DB2 C precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- EXEC SQL INCLUDE payroll;

If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the C precompiler searches for payroll.sqc, then payroll.h, in each directory in which it looks. On UNIX operating systems, the C++ precompiler searches for payroll.sqc, then payroll.sqx, then payroll.hpp, then payroll.h in each directory in which it looks. On OS/2 or Windows-32 bit operating systems, the C++ precompiler searches for payroll.sqx, then payroll.hpp, then payroll.h in each directory in which it looks.

- EXEC SQL INCLUDE 'pay/payroll.h';

If the file name is enclosed in quotation marks, as above, no extension is added to the name.

If the file name in quotation marks does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, on UNIX based systems, if DB2INCLUDE is set to '/disk2:myfiles/c', the C/C++ precompiler searches for './pay/payroll.h', then '/disk2/pay/payroll.h', and finally './myfiles/c/pay/payroll.h'. The path where the file is actually found is displayed in the precompiler messages. On OS/2 and Windows-based operating systems, substitute back slashes (\) for the forward slashes in the above example.

**Note:** The setting of DB2INCLUDE is cached by the DB2 Command Line Processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

To help relate compiler errors back to the original source the precompiler generates ANSI #line macros in the output file. This allows the compiler to report errors using the file name and line number of the source or included source file, rather than the precompiler output.

However, if you specify the PREPROCESSOR option, all the #line macros generated by the precompiler reference the preprocessed file from the external C preprocessor. For more information about the PREPROCESSOR option, see "C Macro Expansion" on page 510.

Some debuggers and other tools that relate source code to object code do not always work well with the #line macro. If the tool you wish to use behaves unexpectedly, use the NOLINEMACRO option (used with DB2 PREP) when precompiling. This will prevent the #line macros from being generated.

---

## Embedding SQL Statements

Embedded SQL statements consist of the following three elements:

| Element               | Correct Syntax          |
|-----------------------|-------------------------|
| Statement initializer | EXEC SQL                |
| Statement string      | Any valid SQL statement |
| Statement terminator  | semicolon (;)           |

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table;
```

The following rules apply to embedded SQL statements:

- You can begin the SQL statement string on the same line as the keyword pair or a separate line. The statement string can be several lines long. Do not split the EXEC SQL keyword pair between lines.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.  
C/C++ comments can be placed before the statement initializer or after the statement terminator.
- Multiple SQL statements and C/C++ statements may be placed on the same line. For example:  

```
EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
```
- The SQL precompiler leaves CR/LFs and TABs in a quoted string as is.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (--) followed by a string of zero or more characters and terminated by a line end. Do not place SQL comments after the SQL statement terminator as they will cause compilation errors because they would appear to be part of the C/C++ language.

You can use comments in a static statement string wherever blanks are allowed. Use the C/C++ comment delimiters /\* \*/, or the SQL comment symbol (--). //-style C++ comments are not permitted within static SQL statements, but they may be used elsewhere in your program. The precompiler removes comments before processing the SQL statement. You

**cannot** use the C and C++ comment delimiters `/* */` or `//` in a dynamic SQL statement. However, you can use them elsewhere in your program.

- You can continue SQL string literals and delimited identifiers over line breaks in C and C++ applications. To do this, use a back slash (`\`) at the end of the line where the break is desired. For example:

```
EXEC SQL SELECT "NA\
ME" INTO :n FROM staff WHERE name='Sa\
nders';
```

Any new line characters (such as carriage return and line feed) are not included in the string or delimited identifier.

- Substitution of white space characters such as end-of-line and TAB characters occur as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a C program. TABs are not modified.

Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, OS/2 uses Carriage Return/Line Feed for end-of-line, whereas UNIX-based systems use just a Line Feed.

---

## Host Variables

Host variables are C or C++ language variables that are referenced within SQL statements. They allow an application to pass input data to and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other C/C++ variable. Obey the rules described in the following sections when naming, declaring, and using host variables.

### Naming Host Variables

The SQL precompiler identifies host variables by their declared name. The following rules apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than SQL, sql, DB2, and db2, which are reserved for system use. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
 char varsql; /* allowed */
 char sqlvar; /* not allowed */
 char SQL_VAR; /* not allowed */
EXEC SQL END DECLARE SECTION;
```

- The precompiler considers host variable names as global to a module. This does not mean, however, that host variables have to be declared as global variables; it is perfectly acceptable to declare host variables as local variables within functions. For example, the following code will work correctly:

```
void f1(int i)
{
 EXEC SQL BEGIN DECLARE SECTION;
 short host_var_1;
 EXEC SQL END DECLARE SECTION;
 EXEC SQL SELECT COL1 INTO :host_var_1 from TBL1;
}
void f2(int i)
{
 EXEC SQL BEGIN DECLARE SECTION;
 short host_var_2;
 EXEC SQL END DECLARE SECTION;
 EXEC SQL INSERT INTO TBL1 VALUES (:host_var_2);
}
```

It is also possible to have several local host variables with the same name as long as they all have the same type and size. To do this, declare the first occurrence of the host variable to the precompiler between BEGIN DECLARE SECTION and END DECLARE SECTION statements, and leave subsequent declarations of the variable out of declare sections. The following code shows an example of this:

```
void f3(int i)
{
 EXEC SQL BEGIN DECLARE SECTION;
 char host_var_3[25];
 EXEC SQL END DECLARE SECTION;
 EXEC SQL SELECT COL2 INTO :host_var_3 FROM TBL2;
}
void f4(int i)
{
 char host_var_3[25];
 EXEC SQL INSERT INTO TBL2 VALUES (:host_var_3);
}
```

Since f3 and f4 are in the same module, and since host\_var\_3 has the same type and length in both functions, a single declaration to the precompiler is sufficient to use it in both places.

## Declaring Host Variables

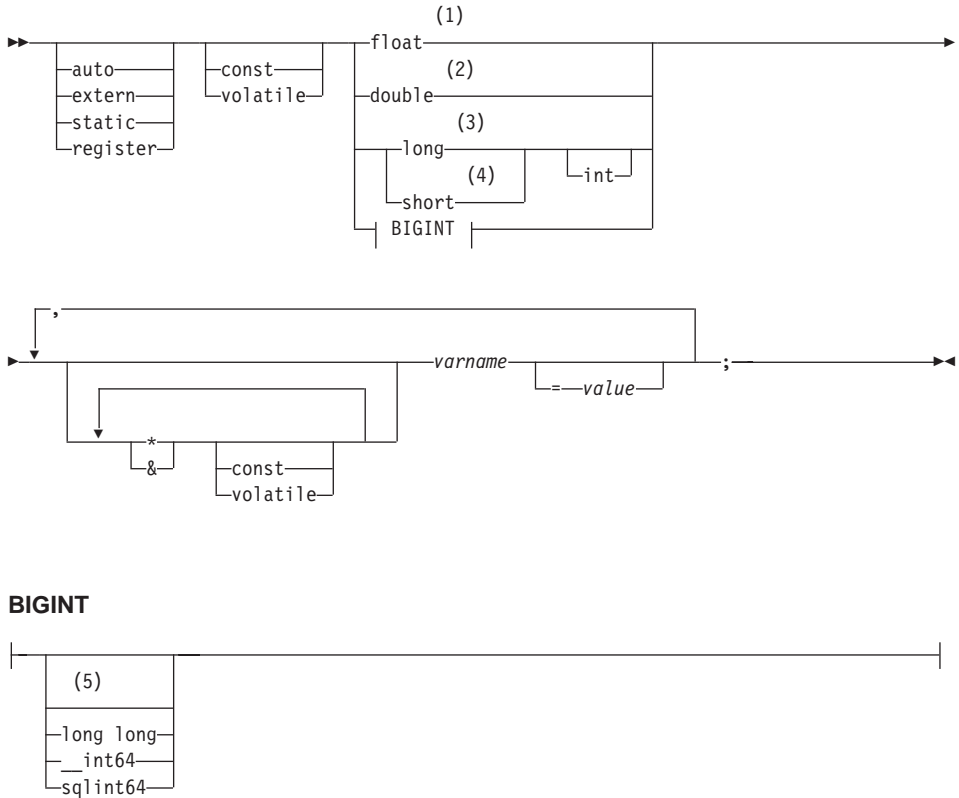
An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The C/C++ precompiler only recognizes a subset of valid C or C++ declarations as valid host variable declarations. These declarations define either numeric or character variables. Typedefs for host variable types are not allowed. Host variables can be grouped into a single host structure. For more information on host structures, see “Host Structure Support” on page 511. You can declare C++ class data members as host variables. For more information on classes, see “Using Class Data Members as Host Variables” on page 517.

A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time or timestamp SQL input or output value. The application must ensure that output variables are long enough to contain the values that they receive.

“Syntax for Numeric Host Variables in C or C++” shows the syntax for declaring numeric host variables in C or C++.

**Syntax for Numeric Host Variables in C or C++**



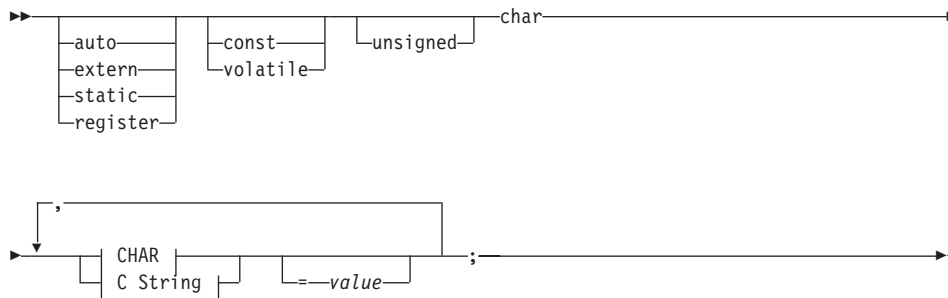


## Notes:

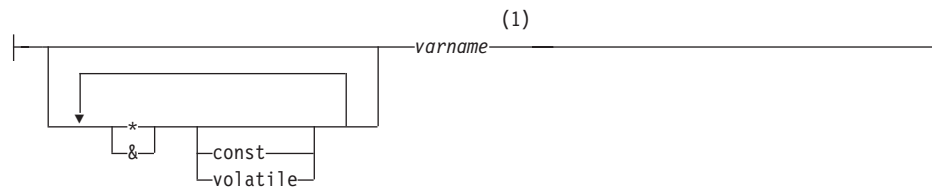
1. REAL (SQLTYPE 480), length 4
2. DOUBLE (SQLTYPE 480), length 8
3. INTEGER (SQLTYPE 496)
4. SMALLINT (SQLTYPE 500)
5. BIGINT (SQLTYPE 492)

“Form 1: Syntax for Fixed and Null-terminated Character Host Variables in C/C++” shows the syntax for declaring fixed and null-terminated character host variables in C or C++.

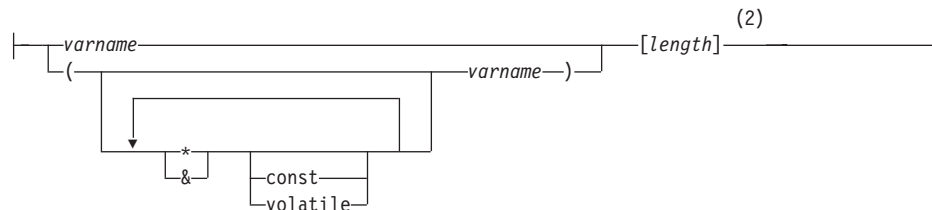
### Form 1: Syntax for Fixed and Null-terminated Character Host Variables in C/C++



### CHAR



### C String

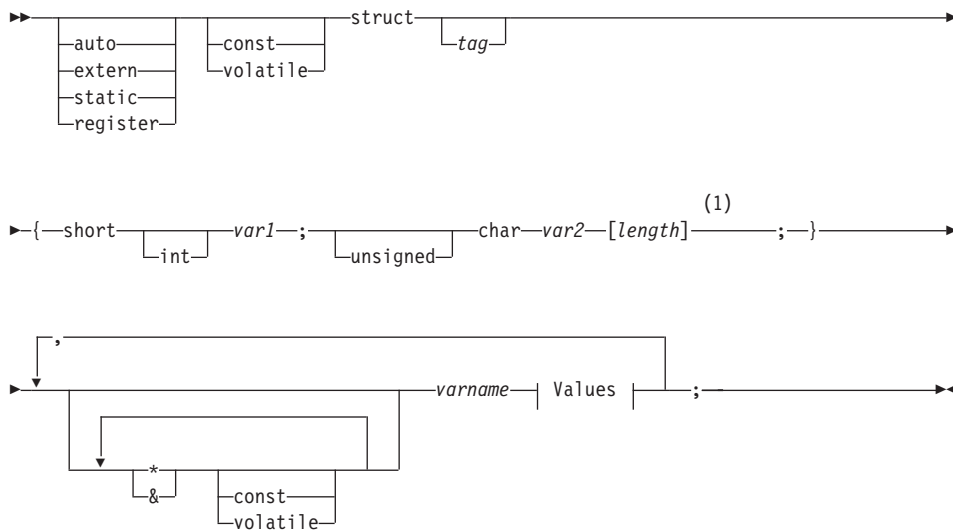


### Notes:

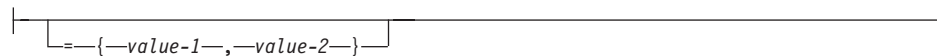
1. CHAR (SQLTYPE 452), Length 1
2. Null-terminated C string (SQLTYPE 460); length can be any valid constant expression

“Form 2: Syntax for Variable Length Character Host Variables in C/C++” shows the syntax for declaring variable length character host variables in C or C++.

### Form 2: Syntax for Variable Length Character Host Variables in C/C++



### Values



### Notes:

1. In **form 2**, *length* can be any valid constant expression. Its value after evaluation determines if the host variable is VARCHAR (SQLTYPE 448) or LONG VARCHAR (SQLTYPE 456).

### Variable Length Character Host Variable Considerations:

1. Although the database manager converts character data to either **form 1** or **form 2** whenever possible, **form 1** corresponds to column types CHAR or VARCHAR while **form 2** corresponds to column types VARCHAR and LONG VARCHAR.

2. If **form 1** is used with a length specifier *[n]*, the value for the length specifier after evaluation must be no greater than 32672, and the string contained by the variable should be null-terminated.
3. If **form 2** is used, the value for the length specifier after evaluation must be no greater than 32 700.
4. In **form 2**, *var1* and *var2* must be simple variable references (no operators), and cannot be used as host variables (*varname* is the host variable).
5. *varname* can be a simple variable name or it can include operators, such as *\*varname*. See “Pointer Data Types” on page 516 for more information.
6. The precompiler determines the SQLTYPE and SQLLEN of all host variables. If a host variable appears in an SQL statement with an indicator variable, the SQLTYPE is assigned to be the base SQLTYPE plus one, for the duration of that statement.
7. The precompiler permits some declarations which are not syntactically valid in C or C++. Refer to your compiler documentation if in doubt of a particular declaration syntax.

## Indicator Variables

Indicator variables should be declared as a short data type.

## Graphic Host Variable Declarations in C or C++

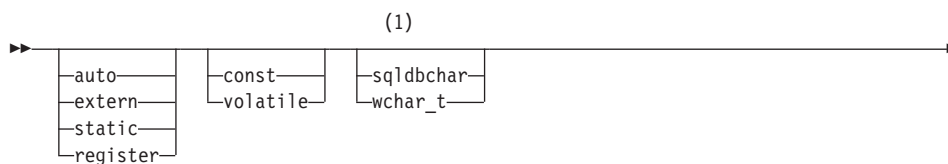
Graphic host variable declarations can take one of three forms:

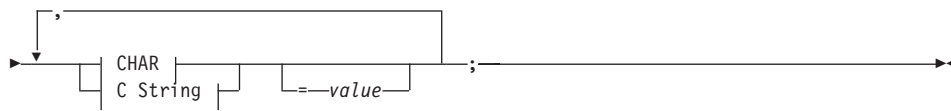
- Single-graphic form
- null-terminated graphic form
- VARGRAPHIC structured form

For details on using graphic host variables, see “Handling Graphic Host Variables” on page 519.

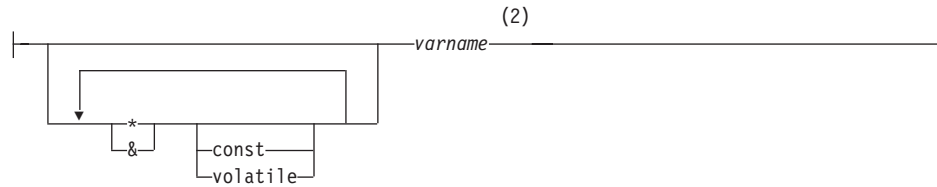
“Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)” shows the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.

### Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)

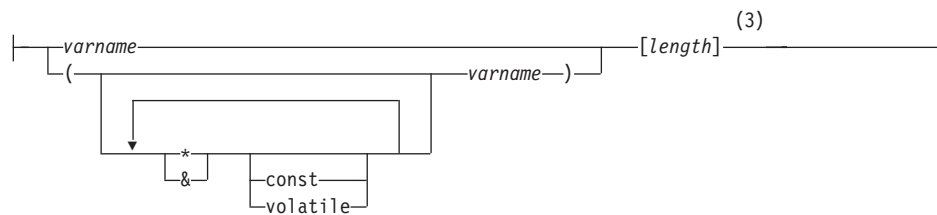




## CHAR



## C String



### Notes:

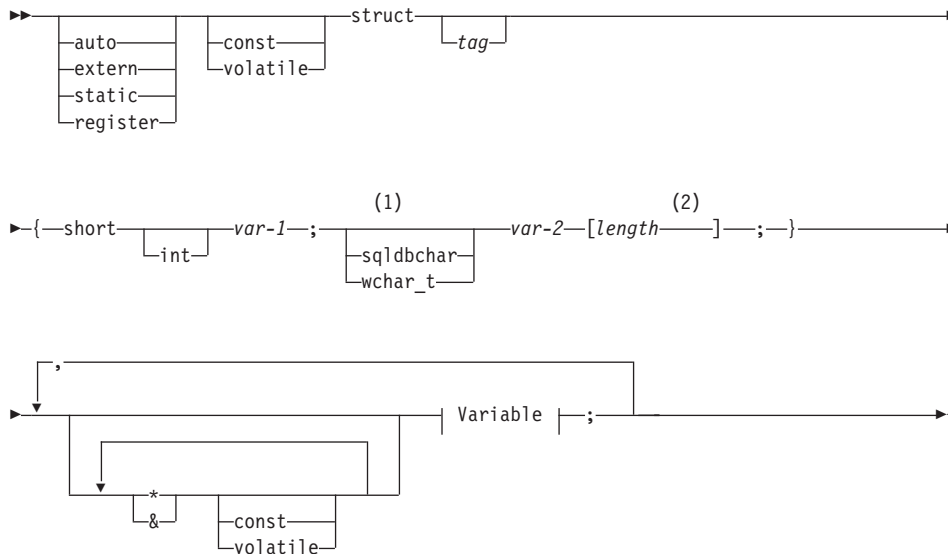
1. To determine which of the two graphic types should be used, see “Selecting the wchar\_t or sqldbchar Data Type” on page 520.
2. GRAPHIC (SQLTYPE 468), Length 1
3. Null-terminated graphic string (SQLTYPE 400)

### Graphic Host Variable Considerations:

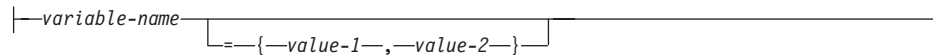
1. The single-graphic form declares a fixed-length graphic string host variable of length 1 with SQLTYPE of 468 or 469.
2. *value* is an initializer. A wide-character string literal (L-literal) should be used if WCHARTYPE CONVERT precompiler option is used.
3. *length* can be any valid constant expression, and its value after evaluation must be greater than or equal to 1 and not greater than the maximum length of VARGRAPHIC, which is 16 336.
4. Null-terminated graphic strings are handled differently depending on the value of the standards level precompile option setting. See “Null-terminated Strings” on page 514 for details.

“Syntax for Graphic Declaration (VARGRAPHIC Structured Form)” shows the syntax for declaring a graphic host variable using the VARGRAPHIC structured form.

#### Syntax for Graphic Declaration (VARGRAPHIC Structured Form)



#### Variable:



#### Notes:

1. To determine which of the two graphic types should be used, see “Selecting the wchar\_t or sqldbchar Data Type” on page 520.
2. *length* can be any valid constant expression. Its value after evaluation determines if the host variable is VARGRAPHIC (SQLTYPE 464) or LONG VARGRAPHIC (SQLTYPE 472). The value of *length* must be greater than or equal to 1 and not greater than the maximum length of LONG VARGRAPHIC which is 16350.

#### Graphic Declaration (VARGRAPHIC Structured Form) Considerations:

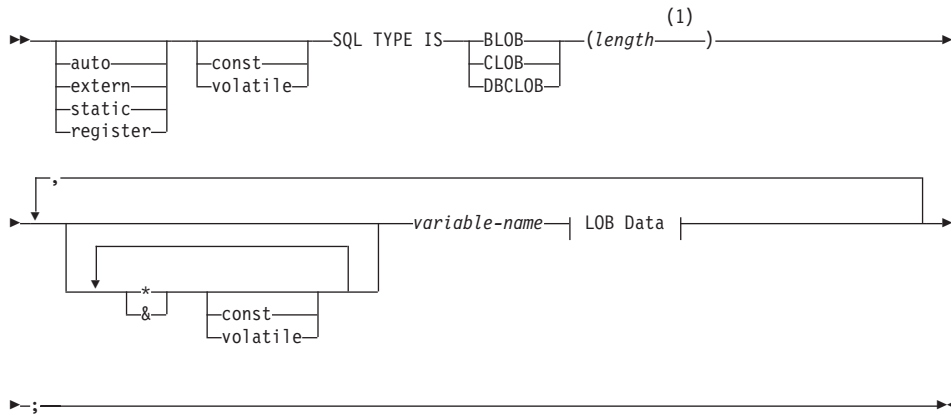
1. *var-1* and *var-2* must be simple variable references (no operators) and cannot be used as host variables.

2. *value-1* and *value-2* are initializers for *var-1* and *var-2*. *value-1* must be an integer and *value-2* should be a wide-character string literal (L-literal) if WCHARTYPE CONVERT precompiler option is used.
3. The struct *tag* can be used to define other data areas, but itself cannot be used as a host variable.

## LOB Data Declarations in C or C++

“Syntax for Large Object (LOB) Host Variables in C/C++” shows the syntax for declaring large object (LOB) host variables in C or C++.

### Syntax for Large Object (LOB) Host Variables in C/C++



### LOB Data



### Notes:

1. *length* can be any valid constant expression, in which the constant K, M, or G can be used. Its value after evaluation must be:
  - for BLOB and CLOB,  $1 \leq \text{length} \leq 2\,147\,483\,647$
  - for DBCLOB,  $1 \leq \text{length} \leq 1\,073\,741\,823$

### LOB Host Variable Considerations:

1. The SQL TYPE IS clause is needed in order to distinguish the three LOB-types from each other so that type-checking and function resolution can be carried out for LOB-type host variables that are passed to functions.

2. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G may be in mixed case.
3. The maximum length allowed for the initialization string, "*init-data*", is 32702 bytes including string delimiters (the same as the existing limit on C/C++ strings within the precompiler).
4. The initialization length, *init-len*, must be a numeric constant (i.e. it cannot include K, M, or G).
5. A length for the LOB must be specified; that is, the following declaration is not permitted:  

```
SQL TYPE IS BLOB my_blob;
```
6. If the LOB is not initialized within the declaration, then no initialization will be done within the precompiler generated code.
7. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

**Note:** Wide character literals, for example, L"Hello", should only be used in a precompiled program if the WCHARTYPE CONVERT precompile option is selected.

8. The precompiler generates a structure tag which can be used to cast to the host variable's type.

#### **BLOB Example:**

Declaration:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

Results in the generation of the following structure:

```
static struct my_blob_t {
 unsigned long length;
 char data[2097152];
} my_blob=SQL_BLOB_INIT("mydata");
```

#### **CLOB Example:**

Declaration:

```
volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

Results in the generation of the following structure:

```
volatile struct var1_t {
 unsigned long length;
 char data[131072000];
} * var1, var2 = {10, "data5data5"};
```

#### **DBCLOB Example:**

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

Precompiled with the WCHARTYPE NOCONVERT option, results in the generation of the following structure:

```
struct my_dbclob1_t {
 unsigned long length;
 sqlbchar data[30000];
} my_dbclob1;
```

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

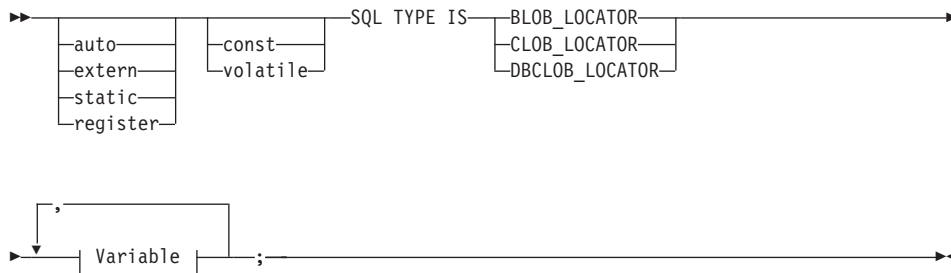
Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

```
struct my_dbclob2_t {
 unsigned long length;
 wchar_t data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

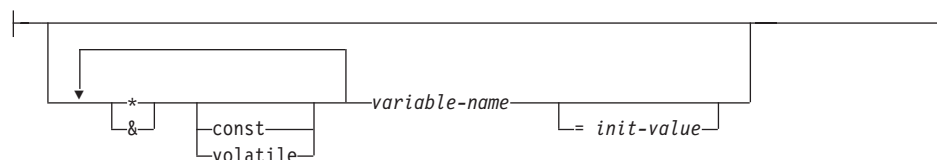
## LOB Locator Declarations in C or C++

“Syntax for Large Object (LOB) Locator Host Variables in C/C++” shows the syntax for declaring large object (LOB) locator host variables in C or C++.

### Syntax for Large Object (LOB) Locator Host Variables in C/C++



### Variable





### LOB Locator Host Variable Considerations:

1. SQL TYPE IS, BLOB\_LOCATOR, CLOB\_LOCATOR, DBCLOB\_LOCATOR may be in mixed case.
2. *init-value* permits the initialization of pointer and reference locator variables. Other types of initialization will have no meaning.

**CLOB Locator Example** (other LOB locator type declarations are similar):

Declaration:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

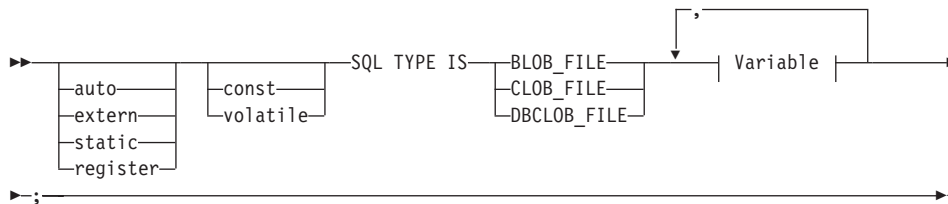
Results in the generation of the following declaration:

```
long my_locator;
```

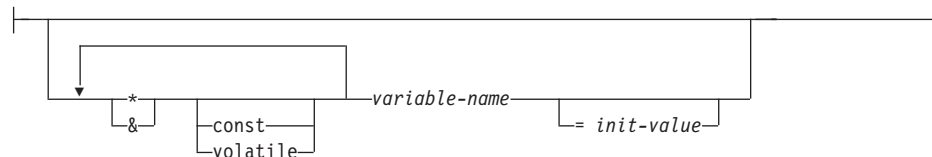
### File Reference Declarations in C or C++

“Syntax for File Reference Host Variables in C/C++” shows the syntax for declaring file reference host variables in C or C++.

#### Syntax for File Reference Host Variables in C/C++



#### Variable



#### Note:

- SQL TYPE IS, BLOB\_FILE, CLOB\_FILE, DBCLOB\_FILE may be in mixed case.

**CLOB File Reference Example** (other LOB file reference type declarations are similar):

Declaration:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

Results in the generation of the following structure:

```
static volatile struct {
 unsigned long name_length;
 unsigned long data_length;
 unsigned long file_options;
 char name[255];
} my_file;
```

## Initializing Host Variables

In C++ declare sections, you cannot initialize host variables using parentheses. The following example shows the correct and incorrect methods of initialization in a declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
 short my_short_2 = 5; /* correct */
 short my_short_1(5); /* incorrect */
EXEC SQL END DECLARE SECTION;
```

## C Macro Expansion

The C/C++ precompiler cannot directly process any C macro used in a declaration within a declare section. Instead, you must first preprocess the source file with an external C preprocessor. To do this, specify the exact command for invoking a C preprocessor to the precompiler through the `PREPROCESSOR` option.

When you specify the `PREPROCESSOR` option, the precompiler first processes all the `SQL INCLUDE` statements by incorporating the contents of all the files referred to in the `SQL INCLUDE` statement into the source file. The precompiler then invokes the external C preprocessor using the command you specify with the modified source file as input. The preprocessed file, which the precompiler always expects to have an extension of ".i", is used as the new source file for the rest of the precompiling process.

Any `#line` macro generated by the precompiler no longer references the original source file, but instead references the preprocessed file. In order to relate any compiler errors back to the original source file, retain comments in the preprocessed file. This helps you to locate various sections of the original source files, including the header files. The option to retain comments is commonly available in C preprocessors, and you can include the option in the command you specify through the `PREPROCESSOR` option. You should not have the C preprocessor output any `#line` macros itself, as they may be incorrectly mixed with ones generated by the precompiler.

### Notes on Using Macro Expansion:

1. The command you specify through the PREPROCESSOR option should include all the desired options but not the name of the input file. For example, for IBM C on AIX you can use the option:  
`x1C -P -DMYMACRO=1`
2. The precompiler expects the command to generate a preprocessed file with a `.i` extension. However, you cannot use redirection to generate the preprocessed file. For example, you **cannot** use the following option to generate a preprocessed file:  
`x1C -E > x.i`
3. Any errors the external C preprocessor encounters are reported in a file with a name corresponding to the original source file but with a `.err` extension.

For example, you can use macro expansion in your source code as follows:

```
#define SIZE 3

EXEC SQL BEGIN DECLARE SECTION;
char a[SIZE+1];
char b[(SIZE+1)*3];
struct
{
 short length;
 char data[SIZE*6];
} m;
SQL TYPE IS BLOB(SIZE+1) x;
SQL TYPE IS CLOB((SIZE+2)*3) y;
SQL TYPE IS DBCLOB(SIZE*2K) z;
EXEC SQL END DECLARE SECTION;
```

The previous declarations resolve to the following after you use the PREPROCESSOR option:

```
EXEC SQL BEGIN DECLARE SECTION;
char a[4];
char b[12];
struct
{
 short length;
 char data[18];
} m;
SQL TYPE IS BLOB(4) x;
SQL TYPE IS CLOB(15) y;
SQL TYPE IS DBCLOB(6144) z;
EXEC SQL END DECLARE SECTION;
```

### Host Structure Support

With host structure support, the C/C++ precompiler allows host variables to be grouped into a single host structure. This provides a shorthand for

referencing that same set of host variables in an SQL statement. For example, the following host structure can be used to access some of the columns in the STAFF table of the SAMPLE database:

```
struct tag
{
 short id;
 struct
 {
 short length;
 char data[10];
 } name;
 struct
 {
 short years;
 double salary;
 } info;
} staff_record;
```

The fields of a host structure can be any of the valid host variable types. These include all numeric, character, and large object types. Nested host structures are also supported up to 25 levels. In the example above, the field info is a sub-structure, whereas the field name is not, as it represents a VARCHAR field. The same principle applies to LONG VARCHAR, VARGRAPHIC and LONG VARGRAPHIC. Pointer to host structure is also supported.

There are two ways to reference the host variables grouped in a host structure in an SQL statement:

1. The host structure name can be referenced in an SQL statement.

```
EXEC SQL SELECT id, name, years, salary
 INTO :staff_record
 FROM staff
 WHERE id = 10;
```

The precompiler converts the reference to staff\_record into a list, separated by commas, of all the fields declared within the host structure. Each field is qualified with the host structure names of all levels to prevent naming conflicts with other host variables or fields. This is equivalent to the following method.

2. Fully qualified host variable names can be referenced in an SQL statement.

```
EXEC SQL SELECT id, name, years, salary
 INTO :staff_record.id, :staff_record.name,
 :staff_record.info.years, :staff_record.info.salary
 FROM staff
 WHERE id = 10;
```

References to field names must be fully qualified even if there are no other host variables with the same name. Qualified sub-structures can also be

referenced. In the example above, `:staff_record.info` can be used to replace `:staff_record.info.years`, `:staff_record.info.salary`.

Since a reference to a host structure (first example) is equivalent to a comma-separated list of its fields, there are instances where this type of reference may lead to an error. For example:

```
EXEC SQL CONNECT TO :staff_record;
```

Here, the `CONNECT` statement expects a single character-based host variable. By giving a host structure instead, the statement results in a precompile-time error:

```
SQL0087N Host variable "staff_record" is a structure used where structure
references are not permitted.
```

Other uses of host structures, which may cause an `SQL0087N` error to occur, include `PREPARE`, `EXECUTE IMMEDIATE`, `CALL`, indicator variables and `SQLDA` references. Host structures with exactly one field are permitted in such situations, as are references to individual fields (second example).

## Indicator Tables

An indicator table is a collection of indicator variables to be used with a host structure. It must be declared as an array of short integers. For example:

```
short ind_tab[10];
```

The example above declares an indicator table with 10 elements. The following shows the way it can be used in an SQL statement:

```
EXEC SQL SELECT id, name, years, salary
 INTO :staff_record INDICATOR :ind_tab
 FROM staff
 WHERE id = 10;
```

The following lists each host structure field with its corresponding indicator variable in the table:

|                                 |                         |
|---------------------------------|-------------------------|
| <b>staff_record.id</b>          | <code>ind_tab[0]</code> |
| <b>staff_record.name</b>        | <code>ind_tab[1]</code> |
| <b>staff_record.info.years</b>  | <code>ind_tab[2]</code> |
| <b>staff_record.info.salary</b> | <code>ind_tab[3]</code> |

**Note:** An indicator table element, for example `ind_tab[1]`, cannot be referenced individually in an SQL statement. The keyword `INDICATOR` is optional. The number of structure fields and indicators do not have to match; any extra indicators are unused, and any extra fields do not have indicators assigned to them.

A scalar indicator variable can also be used in the place of an indicator table to provide an indicator for the first field of the host structure. This is equivalent to having an indicator table with only 1 element. For example:

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
 INTO :staff_record INDICATOR :scalar_ind
 FROM staff
 WHERE id = 10;
```

If an indicator table is specified along with a host variable instead of a host structure, only the first element of the indicator table, for example `ind_tab[0]`, will be used:

```
EXEC SQL SELECT id
 INTO :staff_record.id INDICATOR :ind_tab
 FROM staff
 WHERE id = 10;
```

If an array of short integers is declared within a host structure:

```
struct tag
{
 short i[2];
} test_record;
```

The array will be expanded into its elements when `test_record` is referenced in an SQL statement making `:test_record` equivalent to `:test_record.i[0]`, `:test_record.i[1]`.

## Null-terminated Strings

C/C++ null-terminated strings have their own SQLTYPE (460/461 for character and 468/469 for graphic).

C/C++ null-terminated strings are handled differently depending on the value of the `LANGLEVEL` precompiler option. If a host variable of one of these SQLTYPES and declared length  $n$  is specified within an SQL statement, and the number of bytes (for character types) or double-byte characters (for graphic types) of data is  $k$ , then:

- If the `LANGLEVEL` option on the `PREP` command is `SAA1` (the default):

### For Output:

| If...   | Then...                                                                                                                                                                                              |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $k > n$ | $n$ characters are moved to the target host variable, <code>SQLWARN1</code> is set to 'W', <code>SQLCODE 0</code> ( <code>SQLSTATE 01004</code> ). No null-terminator is placed in the string. If an |

indicator variable was specified with the host variable, the value of the indicator variable is set to  $k$ .

$k = n$   $k$  characters are moved to the target host variable and SQLWARN1 is set to 'N', and SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.

$k < n$   $k$  characters are moved to the target host variable and a null character is placed in character  $k + 1$ . If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.

**For Input:** When the database manager encounters an input host variable of one of these SQLTYPES that does not end with a null-terminator, it will assume that character  $n+1$  will contain the null-terminator character.

- If the LANGLEVEL option on the PREP command is MIA:

**For Output:**

| <b>If...</b> | <b>Then...</b>                                                                                                                                                                                                                                                                                           |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $k \geq n$   | $n - 1$ characters are moved to the target host variable, SQLWARN1 is set to 'W', and SQLCODE 0 (SQLSTATE 01501). The $n$ th character is set to the null-terminator. If an indicator variable was specified with the host variable, the value of the indicator variable is set to $k$ .                 |
| $k + 1 = n$  | $k$ characters are moved to the target host variable, and the null-terminator is placed in character $n$ . If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0.                                                                               |
| $k + 1 < n$  | $k$ characters are moved to the target host variable, $n - k - 1$ blanks are appended on the right starting at character $k + 1$ , then the null-terminator is placed in character $n$ . If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0. |

**For Input:** When the database manager encounters an input host variable of one of these SQLTYPES that does not end with a null character, SQLCODE -302 (SQLSTATE 22501) are returned

When specified in any other SQL context, a host variable of SQLTYPE 460 with length  $n$  is treated as a VARCHAR data type with length  $n$  as defined above. When specified in any other SQL context, a host variable of SQLTYPE 468 with length  $n$  is treated as a VARGRAPHIC data type with length  $n$  as defined above.

## Pointer Data Types

Host variables may be declared as pointers to specific data types with the following restrictions:

- If a host variable is declared as a pointer, then no other host variable may be declared with that same name within the same source file. The following example is not allowed:

```
char mystring[20];
char (*mystring)[20];
```

- Use parentheses when declaring a pointer to a null-terminated character array. In all other cases, parentheses are not allowed. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
char (*arr)[10]; /* correct */
char *(arr); /* incorrect */
char *arr[10]; /* incorrect */
EXEC SQL END DECLARE SECTION;
```

The first declaration is a pointer to a 10-byte character array. This is a valid host variable. The second is an invalid declaration. The parentheses are not allowed in a pointer to a character. The third declaration is an array of pointers. This is not a supported data type.

The host variable declaration:

```
char *ptr
```

is accepted, but it does not mean *null-terminated character string of undetermined length*. Instead, it means a *pointer to a fixed-length, single character host variable*. This may not be what is intended. To define a pointer host variable that can indicate different character strings, use the first declaration form above.

- When pointer host variables are used in SQL statements, they should be prefixed by the same number of asterisks as they were declared with, as in the following example:



```
EXEC SQL BEGIN DECLARE SECTION;
char (*mychar)[20]; /* Pointer to character array of 20 bytes */
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL SELECT column INTO :mychar; /* Correct */
```

- Only the asterisk may be used as an operator over a host variable name.
- The maximum length of a host variable name is not affected by the number of asterisks specified, because asterisks are not considered part of the name.
- Whenever using a pointer variable in an SQL statement, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager.

## Using Class Data Members as Host Variables

You can declare class data members as host variables (but not classes or objects themselves). The following example illustrates the method to use:

```
class STAFF
{
 private:
 EXEC SQL BEGIN DECLARE SECTION;
 char staff_name[20];
 short int staff_id;
 double staff_salary;
 EXEC SQL END DECLARE SECTION;
 short staff_in_db;
 .
};
```

Data members are only directly accessible in SQL statements through the implicit *this* pointer provided by the C++ compiler in class member functions. You **cannot** explicitly qualify an object instance (such as `SELECT name INTO :my_obj.staff_name ...`) in an SQL statement.

If you directly refer to class data members in SQL statements, the database manager resolves the reference using the *this* pointer. For this reason, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager. (This is true whenever pointer host variables are involved in SQL statements.)

The following example shows how you might directly use class data members which you have declared as host variables, in an SQL statement.

```
class STAFF
{
 .
 .
 public:
```

```

 .
 .
 short int hire(void)
 {
 EXEC SQL INSERT INTO staff (name,id,salary)
 VALUES (:staff_name, :staff_id, :staff_salary);
 staff_in_db = (sqlca.sqlcode == 0);
 return sqlca.sqlcode;
 }
};

```

In this example, class data members `staff_name`, `staff_id`, and `staff_salary`, are used directly in the INSERT statement. Because they have been declared as host variables (see the example in “Example of Declaring Class Data Members as Host Variables” on page 517), they are implicitly qualified to the current object with the *this* pointer. In SQL statements, you can also refer to data members that are not accessible through the *this* pointer. You do this by referring to them indirectly using pointer or reference host variables.

The following example shows a new method, *asWellPaidAs* that takes a second object, *otherGuy*. This method references its members indirectly through a local pointer or reference host variable, as you cannot reference its members directly within the SQL statement.

```

short int STAFF::asWellPaidAs(STAFF otherGuy)
{
 EXEC SQL BEGIN DECLARE SECTION;
 short &otherID = otherGuy.staff_id
 double otherSalary;
 EXEC SQL END DECLARE SECTION;
 EXEC SQL SELECT SALARY INTO :otherSalary
 FROM STAFF WHERE id = :otherID;
 if(sqlca.sqlcode == 0)
 return staff_salary >= otherSalary;
 else
 return 0;
}

```

## Using Qualification and Member Operators

You **cannot** use the C++ scope resolution operator `::`, nor the C/C++ member operators `'.'` or `'->'` in embedded SQL statements. You can easily accomplish the same thing through use of local pointer or reference variables, which are set outside the SQL statement to point to the desired scoped variable, and then used inside the SQL statement to refer to it. The following example shows the correct method to use:

```
EXEC SQL BEGIN DECLARE SECTION;
char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
EXEC SQL
 SELECT name INTO :localName FROM STAFF
 WHERE name = 'Sanders';
```

## Handling Graphic Host Variables

To handle graphic data in C or C++ applications, use host variables based on either the `wchar_t` C/C++ data type or the `sqldbcchar` data type provided by DB2. You can assign these types of host variables to columns of a table that are GRAPHIC, VARGRAPHIC, or DBCLOB. For example, you can update or select DBCS data from GRAPHIC or VARGRAPHIC columns of a table.

There are three valid forms for a graphic host variable:

- Single-graphic form.

Single-graphic host variables have an SQLTYPE of 468/469 that is equivalent to GRAPHIC(1) SQL data type. (See “Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)” on page 503.)

- Null-terminated graphic form.

Null-terminated refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). They have an SQLTYPE of 400/401. (See “Syntax for Graphic Declaration (Single-Graphic Form and Null-Terminated Graphic Form)” on page 503.)

- VARGRAPHIC structured form.

VARGRAPHIC structured host variables have an SQLTYPE of 464/465 if their length is between 1 and 16 336 bytes. They have an SQLTYPE of 472/473 if their length is between 2000 and 16 350 bytes. (See “Syntax for Graphic Declaration (VARGRAPHIC Structured Form)” on page 505.)

## Multi-byte Character Encoding

Some character encoding schemes, particularly those from east Asian countries require multiple bytes to represent a character. This external representation of data is called the *multi-byte character code* representation of a character and includes double-byte characters (characters represented by two bytes). Graphic data in DB2 consists of double-byte characters.

To manipulate character strings with double-byte characters, it may be convenient for an application to use an internal representation of data. This internal representation is called the *wide-character code* representation of the double-byte characters and is the format customarily used in the `wchar_t` C/C++ data type. Subroutines that conform to ANSI C and X/OPEN

Portability Guide 4 (XPG4) are available to process wide-character data and to convert data in wide-character format to and from multi-byte format.

Note that although an application can process character data in either multi-byte format or wide-character format, interaction with the database manager is done with DBCS (multi-byte) character codes only. That is, data is stored in and retrieved from GRAPHIC columns in DBCS format. The WCHARTYPE precompiler option is provided to allow application data in wide-character format to be converted to/from multi-byte format when it is exchanged with the database engine.

### Selecting the `wchar_t` or `sqldbchar` Data Type

While the size and encoding of DB2 graphic data is constant from one platform to another for a particular code page, the size and internal format of the ANSI C or C++ `wchar_t` data type depends on which compiler you use and which platform you are on. The `sqldbchar` data type, however, is defined by DB2 to be two bytes in size, and is intended to be a portable way of manipulating DBCS and UCS-2 data in the same format in which it is stored in the database. For more information on UCS-2 data, see “Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations” on page 419 and “Unicode/UCS-2 and UTF-8 Support in DB2 UDB” on page 410.

You can define all DB2 C graphic host variable types using either `wchar_t` or `sqldbchar`. You must use `wchar_t` if you build your application using the WCHARTYPE CONVERT precompile option (as described in “The WCHARTYPE Precompiler Option” on page 521).

**Note:** When specifying the WCHARTYPE CONVERT option on a Windows platform, you should note that `wchar_t` on Windows platforms is Unicode. Therefore, if your C/C++ compiler’s `wchar_t` is not Unicode, the `wcstombs()` function call may fail with SQLCODE -1421 (SQLSTATE=22504). If this happens, you can specify the WCHARTYPE NOCONVERT option, and explicitly call the `wcstombs()` and `mbstowcs()` functions from within your program.

If you build your application with the WCHARTYPE NOCONVERT precompile option, you should use `sqldbchar` for maximum portability between different DB2 client and server platforms. You may use `wchar_t` with WCHARTYPE NOCONVERT, but only on platforms where `wchar_t` is defined as two bytes in length.

If you incorrectly use either `wchar_t` or `sqldbchar` in host variable declarations, you will receive an SQLCODE 15 (no SQLSTATE) at precompile time.

## The WCHARTYPE Precompiler Option

Using the WCHARTYPE precompiler option, you can specify which graphic character format you want to use in your C/C++ application. This option provides you with the flexibility to choose between having your graphic data in multi-byte format or in wide-character format. There are two possible values for the WCHARTYPE option:

### CONVERT

If you select the WCHARTYPE CONVERT option, character codes are converted between the graphic host variable and the database manager. For graphic input host variables, the character code conversion from wide-character format to multi-byte DBCS character format is performed before the data is sent to the database manager, using the ANSI C function `wcstombs()`. For graphic output host variables, the character code conversion from multi-byte DBCS character format to wide-character format is performed before the data received from the database manager is stored in the host variable, using the ANSI C function `mbstowcs()`.

The advantage to using WCHARTYPE CONVERT is that it allows your application to fully exploit the ANSI C mechanisms for dealing with wide-character strings (L-literals, 'wc' string functions, etc.) without having to explicitly convert the data to multi-byte format before communicating with the database manager. The disadvantage is that the implicit conversions may have an impact on the performance of your application at run time, and may increase memory requirements.

If you select WCHARTYPE CONVERT, declare all graphic host variables using `wchar_t` instead of `sqldbchar`.

If you want WCHARTYPE CONVERT behavior, but your application does not need to be precompiled (for example, a CLI application), then define the C preprocessor macro `SQL_WCHART_CONVERT` at compile time. This ensures that certain definitions in the DB2 header files use the data type `wchar_t` instead of `sqldbchar`.

**Note:** The WCHARTYPE CONVERT precompile option is not currently supported in programs running on the DB2 Windows 3.1 client. For those programs, use the default (WCHARTYPE NOCONVERT).

### NOCONVERT (default)

If you choose the WCHARTYPE NOCONVERT option, or do not specify any WCHARTYPE option, no implicit character code conversion occurs between the application and the database manager. Data in a graphic host variable is sent to and received from the database manager as unaltered DBCS characters. This has the

advantage of improved performance, but the disadvantage that your application must either refrain from using wide-character data in `wchar_t` host variables, or must explicitly call the `wcstombs()` and `mbstowcs()` functions to convert the data to and from multi-byte format when interfacing with the database manager.

If you select `WCHARTYPE NOCONVERT`, declare all graphic host variables using the `sqldbchar` type for maximum portability to other DB2 client/server platforms.

Refer to the *Command Reference* for more information.

Other guidelines you need to observe are:

- Since `wchar_t` or `sqldbchar` support is used to handle DBCS data, its use requires DBCS or EUC capable hardware and software. This support is only available in the DBCS environment of DB2 Universal Database, or for dealing with GRAPHIC data in any application (including single-byte applications) connected to a UCS-2 database.
- Non-DBCS characters, and wide-characters which can be converted to non-DBCS characters, should not be used in graphic strings. *Non-DBCS characters* refers to single-byte characters, and non-double byte characters. Graphic strings are not validated to ensure that their values contain only double-byte character code points. Graphic host variables must contain only DBCS data, or, if `WCHARTYPE CONVERT` is in effect, wide-character data which converts to DBCS data. You should store mixed double-byte and single-byte data in character host variables. Note that mixed data host variables are unaffected by the setting of the `WCHARTYPE` option.
- In applications where the `WCHARTYPE NOCONVERT` precompile option is used, L-literals should not be used in conjunction with graphic host variables, since L-literals are in wide-character format. An L-literal is a C wide-character string literal prefixed by the letter L which has the data type "array of `wchar_t`". For example, `L"dbcs-string"` is an L-literal.
- In applications where the `WCHARTYPE CONVERT` precompile option is used, L-literals can be used to initialize `wchar_t` host variables, but cannot be used in SQL statements. Instead of using L-literals, SQL statements should use graphic string constants, which are independent of the `WCHARTYPE` setting.
- The setting of the `WCHARTYPE` option affects graphic data passed to and from the database manager using the `SQLDA` structure as well as host variables. If `WCHARTYPE CONVERT` is in effect, graphic data received from the application through an `SQLDA` will be presumed to be in wide-character format, and will be converted to DBCS format via an implicit call to `wcstombs()`. Similarly, graphic output data received by an application will have been converted to wide-character format before being placed in application storage.

- Not-fenced stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. Ordinary fenced stored procedures may be precompiled with either the CONVERT or NOCONVERT options, which will affect the format of graphic data manipulated by SQL statements contained in the stored procedure. In either case, however, any graphic data passed into the stored procedure through the SQLDA will be in DBCS format. Likewise, data passed out of the stored procedure through the SQLDA must be in DBCS format.
- If an application calls a stored procedure through the Database Application Remote Interface (DARI) interface (the `sqlproc()` API), any graphic data in the input SQLDA must be in DBCS format, or in UCS-2 if connected to a UCS-2 database, regardless of the state of the calling application's WCHARTYPE setting. Likewise, any graphic data in the output SQLDA will be returned in DBCS format, or in UCS-2 if connected to a UCS-2 database, regardless of the WCHARTYPE setting.
- If an application calls a stored procedure through the SQL CALL statement, graphic data conversion will occur on the SQLDA, depending on the calling application's WCHARTYPE setting.
- Graphic data passed to user-defined functions (UDFs) will always be in DBCS format. Likewise, any graphic data returned from a UDF will be assumed to be in DBCS format for DBCS databases, and UCS-2 format for EUC and UCS-2 databases.
- Data stored in DBCLOB files through the use of DBCLOB file reference variables is stored in either DBCS format, or, in the case of UCS-2 databases, in UCS-2 format. Likewise, input data from DBCLOB files is retrieved either in DBCS format, or, in the case of UCS-2 databases, in UCS-2 format.

**Notes:**

1. If you precompile C applications using the WCHARTYPE CONVERT option, DB2 validates the applications' graphic data on both input and output as the data is passed through the conversion functions. If you do **not** use the CONVERT option, no conversion of graphic data, and hence no validation occurs. In a mixed CONVERT/NOCONVERT environment, this may cause problems if invalid graphic data is inserted by a NOCONVERT application and then fetched by a CONVERT application. This data fails the conversion with an SQLCODE -1421 (SQLSTATE 22504) on a FETCH in the CONVERT application.
2. The WCHARTYPE CONVERT precompile option is not currently supported for programs running on the DB2 Windows 3.1 client. In this case, use the default WCHARTYPE NOCONVERT option.



## Japanese or Traditional Chinese EUC, and UCS-2 Considerations

If your application code page is Japanese or Traditional Chinese EUC, or if your application connects to a UCS-2 database, you can access GRAPHIC columns at a database server by using either the CONVERT or the NOCONVERT option, and `wchar_t` or `sqldbchar` graphic host variables, or input/output SQLDAs. In this section, *DBCS format* refers to the UCS-2 encoding scheme for EUC data. Consider the following cases:

- CONVERT option used.

The DB2 client converts graphic data from the wide character format to your application code page, then to UCS-2 before sending the input SQLDA to the database server. Any graphic data is sent to the database server tagged with the UCS-2 code page identifier. Mixed character data is tagged with the application code page identifier. When graphic data is retrieved from a database by a client, it is tagged with the UCS-2 code page identifier. The DB2 client converts the data from UCS-2 to the client application code page, then to the wide character format. If an input SQLDA is used instead of a host variable, then you are required to ensure that graphic data is encoded using the wide character format. This data will be converted to UCS-2 and then sent to the database server. These conversions will impact performance.

- NOCONVERT option used.

The graphic data is assumed by DB2 to be encoded using UCS-2 and is tagged with the UCS-2 code page, and no conversions are done. DB2 assumes that the graphic host variable is being used simply as a bucket. When the NOCONVERT option is chosen, graphic data retrieved from the database server is passed to the application encoded using UCS-2. Any conversions from the application code page to UCS-2 and from UCS-2 to the application code page are your responsibility. Data tagged as UCS-2 is sent to the database server without any conversions or alterations.

To minimize conversions you can either use the NOCONVERT option and handle the conversions in your application, or not use GRAPHIC columns. For the client environments where `wchar_t` encoding is in two-byte Unicode, for example Windows NT or AIX version 4.3 and higher, you can use the NOCONVERT option and work directly with UCS-2. In such cases, your application should handle the difference between big-endian and little-endian architectures. With NOCONVERT option, DB2 Universal Database uses `sqldbchar` which is always two-byte big-endian.

Do not assign IBM-eucJP/IBM-eucTW CS0 (7-bit ASCII) and IBM-eucJP CS2 (Katakana) data to graphic host variables either after conversion to UCS-2 (if NOCONVERT is specified) or by conversion to the wide character format (if CONVERT is specified). This is because characters in both of these EUC code sets become single-byte when converted from UCS-2 to PC DBCS.



In general, although eucJP and eucTW store GRAPHIC data as UCS-2, the GRAPHIC data in these database is still non-ASCII eucJP or eucTW data. Specifically, any space padded to such GRAPHIC data is DBCS space (also known as ideographic space in UCS-2, U+3000). For a UCS-2 database, however, GRAPHIC data can contain any UCS-2 character, and space padding is done with UCS-2 space, U+0020. Keep this difference in mind when you code applications to retrieve UCS-2 data from a UCS-2 database versus UCS-2 data from eucJP and eucTW databases.

For general EUC application development guidelines, see “Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations” on page 419.

## Supported SQL Data Types

Certain predefined C and C++ data types correspond to the database manager column types. Only these C/C++ data types can be declared as host variables.

Table 19 shows the C/C++ equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

**Note:** There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 19. SQL Data Types Mapped to C/C++ Declarations

| SQL Column Type <sup>1</sup>        | C/C++ Data Type                               | SQL Column Type Description                                                                                                  |
|-------------------------------------|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| SMALLINT<br>(500 or 501)            | short<br>short int                            | 16-bit signed integer                                                                                                        |
| INTEGER<br>(496 or 497)             | long<br>long int                              | 32-bit signed integer                                                                                                        |
| BIGINT<br>(492 or 493)              | long long<br>__int64<br>sqlint64 <sup>2</sup> | 64-bit signed integer                                                                                                        |
| REAL <sup>3</sup><br>(480 or 481)   | float                                         | Single-precision floating point                                                                                              |
| DOUBLE <sup>4</sup><br>(480 or 481) | double                                        | Double-precision floating point                                                                                              |
| DECIMAL(p,s)<br>(484 or 485)        | No exact equivalent; use double               | Packed decimal<br><br>(Consider using the CHAR and DECIMAL functions to manipulate packed decimal fields as character data.) |
| CHAR(1)<br>(452 or 453)             | char                                          | Single character                                                                                                             |

Table 19. SQL Data Types Mapped to C/C++ Declarations (continued)

| SQL Column Type <sup>1</sup>                              | C/C++ Data Type                                                                                                         | SQL Column Type Description                                                                          |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| CHAR( <i>n</i> )<br>(452 or 453)                          | No exact equivalent; use<br>char[ <i>n+1</i> ] where <i>n</i> is large enough<br>to hold the data<br>1<= <i>n</i> <=254 | Fixed-length character string                                                                        |
| VARCHAR( <i>n</i> )<br>(448 or 449)                       | struct tag {<br>short int;<br>char[ <i>n</i> ]<br>}                                                                     | Non null-terminated varying character string<br>with 2-byte string length indicator                  |
|                                                           | 1<= <i>n</i> <=32 672                                                                                                   |                                                                                                      |
|                                                           | Alternately use char[ <i>n+1</i> ] where <i>n</i><br>is large enough to hold the data<br>1<= <i>n</i> <=32 672          | null-terminated variable-length character<br>string<br><b>Note:</b> Assigned an SQL type of 460/461. |
| LONG VARCHAR<br>(456 or 457)                              | struct tag {<br>short int;<br>char[ <i>n</i> ]<br>}                                                                     | Non null-terminated varying character string<br>with 2-byte string length indicator                  |
|                                                           | 32 673<= <i>n</i> <=32 700                                                                                              |                                                                                                      |
| CLOB( <i>n</i> )<br>(408 or 409)                          | sql type is<br>clob( <i>n</i> )                                                                                         | Non null-terminated varying character string<br>with 4-byte string length indicator                  |
|                                                           | 1<= <i>n</i> <=2 147 483 647                                                                                            |                                                                                                      |
| CLOB locator variable <sup>5</sup><br>(964 or 965)        | sql type is<br>clob_locator                                                                                             | Identifies CLOB entities residing on the<br>server                                                   |
| CLOB file reference variable <sup>5</sup><br>(808 or 809) | sql type is<br>clob_file                                                                                                | Descriptor for file containing CLOB data                                                             |
| BLOB( <i>n</i> )<br>(404 or 405)                          | sql type is<br>blob( <i>n</i> )                                                                                         | Non null-terminated varying binary string<br>with 4-byte string length indicator                     |
|                                                           | 1<= <i>n</i> <=2 147 483 647                                                                                            |                                                                                                      |
| BLOB locator variable <sup>5</sup><br>(960 or 961)        | sql type is<br>blob_locator                                                                                             | Identifies BLOB entities on the server                                                               |
| BLOB file reference variable <sup>5</sup><br>(804 or 805) | sql type is<br>blob_file                                                                                                | Descriptor for the file containing BLOB data                                                         |
| DATE<br>(384 or 385)                                      | null-terminated character form                                                                                          | Allow at least 11 characters to accommodate<br>the null-terminator.                                  |
|                                                           | VARCHAR structured form                                                                                                 | Allow at least 10 characters.                                                                        |
| TIME<br>(388 or 389)                                      | null-terminated character form                                                                                          | Allow at least 9 characters to accommodate<br>the null-terminator.                                   |
|                                                           | VARCHAR structured form                                                                                                 | Allow at least 8 characters.                                                                         |
| TIMESTAMP<br>(392 or 393)                                 | null-terminated character form                                                                                          | Allow at least 27 characters to accommodate<br>the null-terminator.                                  |
|                                                           | VARCHAR structured form                                                                                                 | Allow at least 26 characters.                                                                        |

Table 19. SQL Data Types Mapped to C/C++ Declarations (continued)

| SQL Column Type <sup>1</sup>                                                                                                                  | C/C++ Data Type                                                                                                                               | SQL Column Type Description                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Note:</b> The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option. |                                                                                                                                               |                                                                                                                                                  |
| GRAPHIC(1)<br>(468 or 469)                                                                                                                    | sqldbchar                                                                                                                                     | Single double-byte character                                                                                                                     |
| GRAPHIC( <i>n</i> )<br>(468 or 469)                                                                                                           | No exact equivalent; use sqldbchar[ <i>n+1</i> ] where <i>n</i> is large enough to hold the data<br>1 ≤ <i>n</i> ≤ 127                        | Fixed-length double-byte character string                                                                                                        |
| VARGRAPHIC( <i>n</i> )<br>(464 or 465)                                                                                                        | struct tag {<br>short int;<br>sqldbchar[ <i>n</i> ]<br>}                                                                                      | Non null-terminated varying double-byte character string with 2-byte string length indicator                                                     |
|                                                                                                                                               | 1 ≤ <i>n</i> ≤ 16 336<br><br>Alternately use sqldbchar[ <i>n+1</i> ] where <i>n</i> is large enough to hold the data<br>1 ≤ <i>n</i> ≤ 16 336 | null-terminated variable-length double-byte character string<br><b>Note:</b> Assigned an SQL type of 400/401.                                    |
| LONG VARGRAPHIC<br>(472 or 473)                                                                                                               | struct tag {<br>short int;<br>sqldbchar[ <i>n</i> ]<br>}                                                                                      | Non null-terminated varying double-byte character string with 2-byte string length indicator                                                     |
|                                                                                                                                               | 16 337 ≤ <i>n</i> ≤ 16 350                                                                                                                    |                                                                                                                                                  |
| <b>Note:</b> The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE CONVERT option.   |                                                                                                                                               |                                                                                                                                                  |
| GRAPHIC(1)<br>(468 or 469)                                                                                                                    | wchar_t                                                                                                                                       | <ul style="list-style-type: none"> <li>• Single wide character (for C-type)</li> <li>• Single double-byte character (for column type)</li> </ul> |
| GRAPHIC( <i>n</i> )<br>(468 or 469)                                                                                                           | No exact equivalent; use wchar_t [n+1] where <i>n</i> is large enough to hold the data<br>1 ≤ <i>n</i> ≤ 127                                  | Fixed-length double-byte character string                                                                                                        |
| VARGRAPHIC( <i>n</i> )<br>(464 or 465)                                                                                                        | struct tag {<br>short int;<br>wchar_t [n]<br>}                                                                                                | Non null-terminated varying double-byte character string with 2-byte string length indicator                                                     |
|                                                                                                                                               | 1 ≤ <i>n</i> ≤ 16 336<br><br>Alternately use char[ <i>n+1</i> ] where <i>n</i> is large enough to hold the data<br>1 ≤ <i>n</i> ≤ 16 336      | null-terminated variable-length double-byte character string<br><b>Note:</b> Assigned an SQL type of 400/401.                                    |

Table 19. SQL Data Types Mapped to C/C++ Declarations (continued)

| SQL Column Type <sup>1</sup>                                                             | C/C++ Data Type                                | SQL Column Type Description                                                                  |
|------------------------------------------------------------------------------------------|------------------------------------------------|----------------------------------------------------------------------------------------------|
| LONG VARCHAR(n)<br>(472 or 473)                                                          | struct tag {<br>short int;<br>wchar_t [n]<br>} | Non null-terminated varying double-byte character string with 2-byte string length indicator |
|                                                                                          | 16 337<=n<=16 350                              |                                                                                              |
| <b>Note:</b> The following data types are only available in the DBCS or EUC environment. |                                                |                                                                                              |
| DBCLOB(n)<br>(412 or 413)                                                                | sql type is<br>dbclob(n)                       | Non null-terminated varying double-byte character string with 4-byte string length indicator |
|                                                                                          | 1<=n<=1 073 741 823                            |                                                                                              |
| DBCLOB locator variable <sup>5</sup><br>(968 or 969)                                     | sql type is<br>dbclob_locator                  | Identifies DBCLOB entities residing on the server                                            |
| DBCLOB file reference<br>variable <sup>5</sup><br>(812 or 813)                           | sql type is<br>dbclob_file                     | Descriptor for file containing DBCLOB data                                                   |

**Notes:**

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. Windows operating systems use \_\_int64, where \_\_ represents 2 underscores. For platform compatibility, use sqlint64. The DB2 Universal Database sqlsystem.h header file will type define sqlint64 as "\_\_int64" on the Windows NT platform when using the Microsoft compiler; otherwise, sqlint64 will be type defined as "long long".
3. FLOAT(n) where 0 < n < 25 is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
4. The following SQL types are synonyms for DOUBLE:
  - FLOAT
  - FLOAT(n) where 24 < n < 54 is
  - DOUBLE PRECISION
5. This is not a column type but a host variable type.

The following is a sample SQL declare section with host variables declared for supported SQL data types.

```
EXEC SQL BEGIN DECLARE SECTION;
...
 short age = 26; /* SQL type 500 */
 short year; /* SQL type 500 */
 long salary; /* SQL type 496 */
 long deptno; /* SQL type 496 */
 float bonus; /* SQL type 480 */
 double wage; /* SQL type 480 */
 char mi; /* SQL type 452 */
 char name[6]; /* SQL type 460 */
 struct {
 short len;
```

```

 char data[24];
 } address; /* SQL type 448 */
struct {
 short len;
 char data[5764];
 } voice; /* SQL type 456 */
sql type is clob(1m)
 chapter; /* SQL type 408 */
sql type is clob_locator
 chapter_locator; /* SQL type 964 */
sql type is clob_file
 chapter_file_ref; /* SQL type 808 */
sql type is blob(1m)
 video; /* SQL type 404 */
sql type is blob_locator
 video_locator; /* SQL type 960 */
sql type is blob_file
 video_file_ref; /* SQL type 804 */
sql type is dbclob(1m)
 tokyo_phone_dir; /* SQL type 412 */
sql type is dbclob_locator
 tokyo_phone_dir_lctr; /* SQL type 968 */
sql type is dbclob_file
 tokyo_phone_dir_flref; /* SQL type 812 */
struct {
 short len;
 sqldbchar data[100];
 } vargraphic1; /* SQL type 464 */
/* Precompiled with
WCHARTYPE NOCONVERT option */
struct {
 short len;
 wchar_t data[100];
 } vargraphic2; /* SQL type 464 */
/* Precompiled with
WCHARTYPE CONVERT option */
struct {
 short len;
 sqldbchar data[10000];
 } long_vargraphic1; /* SQL type 472 */
/* Precompiled with
WCHARTYPE NOCONVERT option */
struct {
 short len;
 wchar_t data[10000];
 } long_vargraphic2; /* SQL type 472 */
/* Precompiled with
WCHARTYPE CONVERT option */
sqldbchar graphic1[100]; /* SQL type 468 */
/* Precompiled with
WCHARTYPE NOCONVERT option */
wchar_t graphic2[100]; /* SQL type 468 */
/* Precompiled with
WCHARTYPE CONVERT option */
char date[11]; /* SQL type 384 */

```

```

char time[9]; /* SQL type 388 */
char timestamp[27]; /* SQL type 392 */
short wage_ind; /* Null indicator */
...
EXEC SQL END DECLARE SECTION;

```

The following are additional rules for supported C/C++ data types:

- The data type `char` can be declared as `char` or `unsigned char`.
- The database manager processes null-terminated variable-length character string data type `char[n]` (data type 460), as `VARCHAR(m)`.
  - If `LANGLEVEL` is `SAA1`, the host variable length  $m$  equals the character string length  $n$  in `char[n]` or the number of bytes preceding the first null-terminator (`\0`), whichever is smaller.
  - If `LANGLEVEL` is `MIA`, the host variable length  $m$  equals the number of bytes preceding the first null-terminator (`\0`).
- The database manager processes null-terminated, variable-length graphic string data type, `wchar_t[n]` or `sqldbcchar[n]` (data type 400), as `VARGRAPHIC(m)`.
  - If `LANGLEVEL` is `SAA1`, the host variable length  $m$  equals the character string length  $n$  in `wchar_t[n]` or `sqldbcchar[n]`, or the number of characters preceding the first graphic null-terminator, whichever is smaller.
  - If `LANGLEVEL` is `MIA`, the host variable length  $m$  equals the number of characters preceding the first graphic null-terminator.
- Unsigned numeric data types are not supported.
- The C/C++ data type `int` is not allowed since its internal representation is machine dependent.

## FOR BIT DATA

The standard C or C++ string type 460 should not be used for columns designated FOR BIT DATA. The database manager truncates this data type when a null character is encountered. Use either the `VARCHAR` (SQL type 448) or `CLOB` (SQL type 408) structures.

---

## SQLSTATE and SQLCODE Variables

When using the `LANGLEVEL` precompile option with a value of `SQL92E`, the following two declarations may be included as host variables:

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6]
long SQLCODE;

```

```
.
. .
EXEC SQL END DECLARE SECTION;
```

If neither of these is specified, the `SQLCODE` declaration is assumed during the precompile step. Note that when using this option, the `INCLUDE SQLCA` statement should not be specified.

In an application that is made up of multiple source files, the `SQLCODE` and `SQLSTATE` variables may be defined in the first source file as above. Subsequent source files should modify the definitions as follows:

```
extern long SQLCODE;
extern char SQLSTATE[6];
```





---

## Chapter 17. Programming in Java

|                                           |     |                                           |     |
|-------------------------------------------|-----|-------------------------------------------|-----|
| Programming Considerations . . . . .      | 533 | Embedding SQL Statements . . . . .        | 548 |
| Comparison of SQLJ to JDBC . . . . .      | 533 | Declaring Iterators for Positioned        |     |
| Advantages Over Other Languages . . . . . | 534 | Updates and Deletes . . . . .             | 549 |
| SQL Security . . . . .                    | 534 | SQLJ Example: App.sqlj . . . . .          | 550 |
| Source and Output Files . . . . .         | 534 | Host Variables. . . . .                   | 554 |
| Java Class Libraries . . . . .            | 535 | Calls to Stored Procedures and Functions  | 554 |
| Java Packages . . . . .                   | 535 | Compiling and Running SQLJ Programs       | 555 |
| Supported SQL Data Types . . . . .        | 535 | SQLJ Precompile Options . . . . .         | 556 |
| SQLSTATE and SQLCODE Values . . . . .     | 537 | Stored Procedures and UDFs. . . . .       | 557 |
| DB2 Trace Facilities . . . . .            | 538 | Where to Put Java Classes. . . . .        | 558 |
| Creating Java Applications and Applets    | 538 | Debugging Stored Procedures . . . . .     | 559 |
| How Does It Work? . . . . .               | 538 | Preparing to Debug . . . . .              | 559 |
| JDBC Programming . . . . .                | 540 | Populating the Debug Table . . . . .      | 560 |
| How the DB2Appl Program Works. . . . .    | 541 | Invoking the Debugger. . . . .            | 562 |
| JDBC Example: DB2Appl.java . . . . .      | 542 | Java Stored Procedures and UDFs . . . . . | 562 |
| Distributing a JDBC Application . . . . . | 543 | Installing, Replacing, and Removing       |     |
| Distributing and Running a JDBC Applet    | 543 | JAR Files . . . . .                       | 563 |
| JDBC 2.0 . . . . .                        | 544 | Function Definitions. . . . .             | 564 |
| Features. . . . .                         | 544 | Using LOBs and Graphical Objects With     |     |
| Compatibility . . . . .                   | 544 | JDBC 1.22 . . . . .                       | 566 |
| Performing the Installation . . . . .     | 545 | JDBC and SQLJ Interoperability . . . . .  | 567 |
| SQLJ Programming . . . . .                | 545 | Session Sharing . . . . .                 | 567 |
| DB2 SQLJ Support . . . . .                | 546 | Connection Resource Management . . . . .  | 567 |
| DB2 SQLJ Restrictions . . . . .           | 547 |                                           |     |

---

### Programming Considerations

DB2 Universal Database implements two standards-based Java programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLJ). This chapter provides an overview of JDBC and SQLJ programming, but focuses on the aspects specific to DB2. See the DB2 Universal Database Java website at <http://www.software.ibm.com/data/db2/java/> for links to the JDBC and SQLJ specifications.

### Comparison of SQLJ to JDBC

The JDBC API allows you to write Java programs that make dynamic SQL calls to databases. SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. You must translate a SQLJ source file with the SQLJ translator before you can compile the resulting Java source code.

For information on building JDBC and SQLJ applications, refer to the *Application Building Guide*.

## Advantages Over Other Languages

Programming languages containing embedded SQL are called host languages. Java differs from the traditional host languages C, COBOL, and FORTRAN, in ways that significantly affect how it embeds SQL:

- SQLJ and JDBC are open standards, enabling you to easily port SQLJ or JDBC applications from other standards-compliant database systems to DB2 Universal Database.
- All Java types representing composite data, and data of varying sizes, have a distinguished value, `null`, which can be used to represent the SQL NULL state, giving Java programs an alternative to NULL indicators that are a fixture of other host languages.
- Java is designed to support programs that are automatically heterogeneously portable (also called "super portable" or simply "downloadable"). Along with Java's type system of classes and interfaces, this feature enables component software. In particular, an SQLJ translator written in Java can call components that are specialized by database vendors in order to leverage existing database functions such as authorization, schema checking, type checking, transactional, and recovery capabilities, and to generate code optimized for specific databases.
- Java is designed for binary portability in heterogeneous networks, which promises to enable binary portability for database applications that use static SQL.

## SQL Security

By default, a JDBC program executes SQL statements with the privileges assigned to the person who runs the program. In contrast, an SQLJ program executes SQL statements with the privileges assigned to the person who created the database package.

## Source and Output Files

Source files have the following extensions:

- .java** Java source files, which require no precompiling. You can compile these files with the `javac` Java compiler included with your Java development environment.
- .sqlj** SQLJ source files, which require translation with the `sqlj` translator. The translator creates:
  - one `.class` bytecode file
  - one `.ser` profile file per connection context

The corresponding output files have the following extensions:

- .class** JDBC and SQLJ bytecode compiled files.
- .ser** SQLJ serialized profile files. You create packages in the database for each profile file with the db2profc utility.

For an example of how to compile and run an SQLJ program, see “Compiling and Running SQLJ Programs” on page 555.

## Java Class Libraries

DB2 Universal Database provides class libraries for JDBC and SQLJ support, which you must provide in your CLASSPATH or include with your applets as follows:

### **db2java.zip**

Provides the JDBC driver and JDBC and SQLJ support classes, including stored procedure and UDF support.

### **sqlj.zip**

Provides the SQLJ translator class files.

### **runtime.zip**

Provides Java run-time support for SQLJ applications and applets.

## Java Packages

To use the class libraries included with DB2 in your own applications, you must include the appropriate `import package` statements at the top of your source files. You can use the following packages in your Java applications:

### **java.sql.\***

The JDBC API included in your JDK. You must import this package in every JDBC and SQLJ program.

### **sqlj.runtime.\***

SQLJ support included with DB2 Software Developer's Kit. You must import this package in every SQLJ program.

### **sqlj.runtime.ref.\***

SQLJ support included with DB2 Software Developer's Kit. You must import this package in every SQLJ program.

## Supported SQL Data Types

Table 20 on page 536 shows the Java equivalent of each SQL data type, based on the JDBC specification for data type mappings. Note that some mappings depend on whether you use the JDBC version 1.22 or version 2.0 driver. The JDBC driver converts the data exchanged between the application and the

database using the following mapping schema. Use these mappings in your Java applications and your PARAMETER STYLE JAVA stored procedures and UDFs. For information on data type mappings for PARAMETER STYLE DB2GENERAL stored procedures and UDFs, see “Supported SQL Data Types” on page 659.

**Note:** There is no host variable support for the DATALINK data type in any of the programming languages supported by DB2.

Table 20. SQL Data Types Mapped to Java Declarations

| SQL Column Type                       | Java Data Type                               | SQL Column Type Description                                                      |
|---------------------------------------|----------------------------------------------|----------------------------------------------------------------------------------|
| SMALLINT<br>(500 or 501)              | short                                        | 16-bit, signed integer                                                           |
| INTEGER<br>(496 or 497)               | int                                          | 32-bit, signed integer                                                           |
| BIGINT<br>(492 or 493)                | long                                         | 64-bit, signed integer                                                           |
| REAL<br>(480 or 481)                  | float                                        | Single precision floating point                                                  |
| DOUBLE<br>(480 or 481)                | double                                       | Double precision floating point                                                  |
| DECIMAL( <i>p,s</i> )<br>(484 or 485) | java.math.BigDecimal                         | Packed decimal                                                                   |
| CHAR( <i>n</i> )<br>(452 or 453)      | String                                       | Fixed-length character string of length <i>n</i> where <i>n</i> is from 1 to 254 |
| VARCHAR( <i>n</i> )<br>(448 or 449)   | String                                       | Variable-length character string                                                 |
| LONG VARCHAR<br>(456 or 457)          | String                                       | Long variable-length character string                                            |
| CHAR( <i>n</i> )<br>FOR BIT DATA      | byte[]                                       | Fixed-length character string of length <i>n</i> where <i>n</i> is from 1 to 254 |
| VARCHAR( <i>n</i> )<br>FOR BIT DATA   | byte[]                                       | Variable-length character string                                                 |
| LONG VARCHAR<br>FOR BIT DATA          | byte[]                                       | Long variable-length character string                                            |
| BLOB( <i>n</i> )<br>(404 or 405)      | JDBC 1.22: byte[]<br>JDBC 2.0: java.sql.Blob | Large object variable-length binary string                                       |
| CLOB( <i>n</i> )<br>(408 or 409)      | JDBC 1.22: String<br>JDBC 2.0: java.sql.Clob | Large object variable-length character string                                    |
| DBCLOB( <i>n</i> )<br>(412 or 413)    | JDBC 1.22: String<br>JDBC 2.0: java.sql.Clob | Large object variable-length double-byte character string                        |

Table 20. SQL Data Types Mapped to Java Declarations (continued)

| SQL Column Type           | Java Data Type     | SQL Column Type Description |
|---------------------------|--------------------|-----------------------------|
| DATE<br>(384 or 385)      | java.sql.Date      | 10-byte character string    |
| TIME<br>(388 or 389)      | java.sql.Time      | 8-byte character string     |
| TIMESTAMP<br>(392 or 393) | java.sql.Timestamp | 26-byte character string    |

## SQLSTATE and SQLCODE Values

While host languages like C/C++ and COBOL use a CHECKERR utility to determine the success of an SQL statement, JDBC and SQLJ programs throw an SQLException if an SQL error occurs. A String object available through the SQLException.getSQLState() instance method represents SQLSTATE. An int variable accessible through the SQLException.getErrorCode() instance method represents the SQLCODE. You can access the corresponding SQL message as a Java String object through the SQLException.getMessage() instance method.

For example:

```
int sqlCode=0; // Variable to hold SQLCODE
String sqlState=0; // Variable to hold SQLSTATE

try
{
 // JDBC statements may throw SQLExceptions
 stmt.executeQuery("Your JDBC statement here");

 // SQLJ statements may also throw SQLExceptions
 #sql {..... your SQLJ statement here};
}

/* Here's how you can check for SQLCODEs and SQLSTATE */

catch (SQLException e)
{
 sqlCode = e.getErrorCode() // Get SQLCODE
 sqlState = e.getSQLState() // Get SQLSTATE

 if (sqlState.equals("-1234"))
 {
 // Your code here to handle -1234 SQLSTATE
 }
 else
 {
 // Your code here to handle other errors
 }
}
```

```

 }
 System.err.println(e.getMessage()) ; // Print the exception
 System.exit(1); // Exit
}

```

## DB2 Trace Facilities

SQLJ programs access DB2 using the DB2 JDBC driver, which in turn uses the DB2 CLI/ODBC driver. Therefore, both the CLI/ODBC/JDBC trace facility and the DB2 trace facility, db2trc, can be used to diagnose problems. Details on how to take the above traces are explained in the *Troubleshooting Guide*.

You can install run-time call tracing capability into SQLJ programs. The utility operates on the profiles associated with a program. Suppose a program uses a profile called App\_SJProfile0. To install call tracing into the program, use the command:

```
profdb App_SJProfile0.ser
```

The profdb utility uses the Java Virtual Machine to run the main() method of class sqlj.runtime.profile.util.AuditorInstaller. For more details on usage and options for the AuditorInstaller class, visit the DB2 Java website at <http://www.software.ibm.com/data/db2/java>.

## Creating Java Applications and Applets

Whether your application or applet uses JDBC or SQLJ, you need to familiarize yourself with the JDBC specification *JDBC: A Java SQL API*, which is available from Sun Microsystems. See the DB2 Java website at <http://www.software.ibm.com/data/db2/java/> for links to JDBC and SQLJ resources. This specification describes how to call JDBC APIs to access a database and manipulate data in that database.

You should also read through this section to learn about DB2's extensions to JDBC and its few limitations (see "JDBC 2.0" on page 544). If you plan to create UDFs or stored procedures in Java, see "Creating and Using Java User-Defined Functions" on page 312 and "Java Stored Procedures and UDFs" on page 562, as there are considerations that are different for Java than for other languages.

To build and run JDBC and SQLJ applications and applets, you must set up your operating system environment according to the instructions in the *Application Building Guide*.

### How Does It Work?

DB2's Java enablement has three independent components:

- Support for client applications and applets written in Java using JDBC to access DB2 (see “JDBC Programming” on page 540)
- Precompile and binding support for client applications and applets written in Java using SQLJ to access DB2 (see “SQLJ Programming” on page 545)
- Support for Java UDFs and stored procedures on the server (see “Stored Procedures and UDFs” on page 557)

**Application Support:** Figure 9 illustrates how a DB2 JDBC application works. You can think of a DB2 JDBC application as a DB2 CLI application, only you write it using the Java language. Calls to JDBC are translated to calls to DB2 CLI through Java native methods. JDBC requests flow from the DB2 client through DB2 CLI to the DB2 server.

SQLJ applications use this JDBC support, and in addition require the SQLJ run-time classes to authenticate and execute any SQL packages that were bound to the database at the precompiling and binding stage.

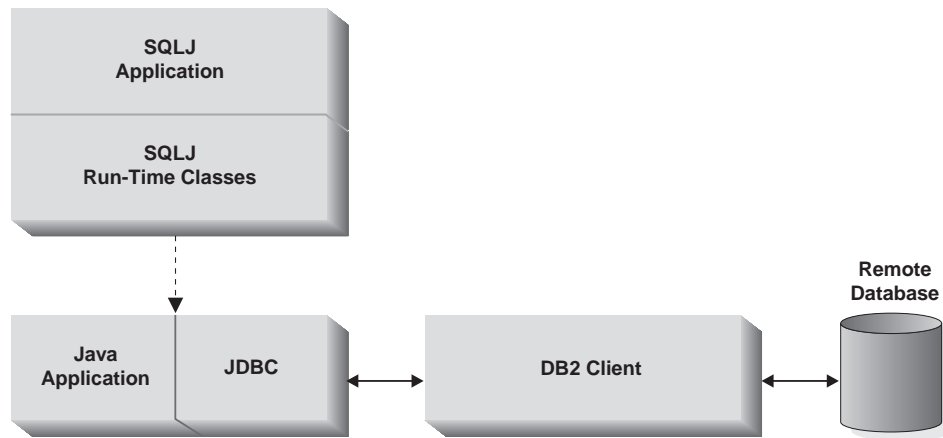


Figure 9. DB2's Java Application Implementation

**Applet Support:** Figure 10 on page 540 illustrates how the JDBC applet driver works. The driver consists of a JDBC client and a JDBC server. The JDBC client driver is loaded on the Web browser along with the applet. When the applet requests a connection to a DB2 database, the client opens a TCP/IP socket to the JDBC server on the machine where the Web server is running. After a connection is set up, the client sends each of the subsequent database access requests from the applet to the JDBC server through the TCP/IP connection. The JDBC server then makes corresponding CLI (ODBC) calls to perform the task. Upon completion, the JDBC server sends the results back to the client through the connection.

SQLJ applets add the SQLJ client driver on top of the JDBC client driver, but otherwise work the same as JDBC applets.

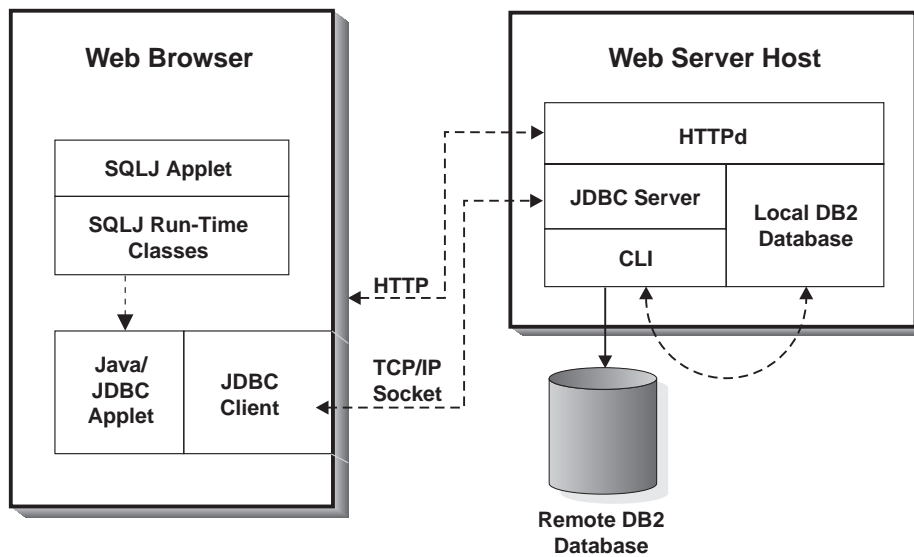


Figure 10. DB2 Java Applet Implementation

## JDBC Programming

Whether you're writing an application or applet, you typically do the following:

1. Import the appropriate Java packages and classes (`java.sql.*`)
2. Load the appropriate JDBC driver (`COM.ibm.db2.jdbc.app.DB2Driver` for applications; `COM.ibm.db2.jdbc.net.DB2Driver` for applets)
3. Connect to the database, specifying the location with a URL (as defined in Sun's JDBC specification) and using the `db2` subprotocol. For applets, you must also provide the `userid`, `password`, `host name`, and the `port number` for the applet server; for applications, the DB2 client provides the required values.
4. Pass SQL statements to the database
5. Receive the results
6. Close the connection

After coding your program, compile it as you would any other Java program. You don't need to perform any special precompile or bind steps.



## How the DB2Appl Program Works

The following sample program, DB2Appl.java, demonstrates how to code a JDBC program for DB2.

1. **Import the JDBC package.** Every JDBC and SQLJ program must import the JDBC package.
2. **Declare a Connection object.** The Connection object establishes and manages the database connection.
3. **Set database URL variable.** The DB2 application driver accepts URLs that take the form of `jdbc:db2:database name`.
4. **Connect to database.** The `DriverManager.getConnection()` method is most often used with the following parameters:

**`getConnection(String url)`**

Establish a connection to the database specified by *url* with the default username and password.

**`getConnection(String url, String userid, String password)`**

Establish a connection to the database specified by *url* with the values for username and password specified by *userid* and *passwd* respectively.

5. **Create a Statement object.** Statement objects send SQL statements to the database.
6. **Execute an SQL SELECT statement.** Use the `executeQuery()` method for SQL statements, like SELECT statements, that return a single result set. Assign the result to a `ResultSet` object.
7. **Retrieve rows from the ResultSet.** The `ResultSet` object allows you to treat a result set like a cursor in host language embedded SQL. The `ResultSet.next()` method moves the cursor to the next row and returns Boolean false if the final row of the result set has been reached. Restrictions on result set processing depend on the level of the JDBC API that is enabled through the database manager configuration parameters.
  - The JDBC 2.0 API allows you to scroll backwards and forwards through a result set, inserting, updating, and deleting rows.
  - The JDBC 1.2 API restricts you to scrolling forward through a read-only result set with the `ResultSet.next()` method.
8. **Return the value of a column.** The `ResultSet.getString(n)` returns the value of the *n*<sup>th</sup> column as a String object.
9. **Execute an SQL UPDATE statement.** Use the `executeUpdate()` method for SQL UPDATE statements. The method returns the number of rows updated as an int value.

### JDBC Example: DB2Appl.java

```
import java.sql.*; 1

class DB2Appl {

 static {
 try {
 Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
 } catch (Exception e) {
 System.out.println(e);
 }
 }

 public static void main(String argv[]) {
 Connection con = null; 2

 // URL is jdbc:db2:dbname
 String url = "jdbc:db2:sample"; 3

 try {
 if (argv.length == 0) {
 // connect with default id/password
 con = DriverManager.getConnection(url);
 }
 else if (argv.length == 2) {
 String userid = argv[0];
 String passwd = argv[1];

 // connect with user-provided username and password
 con = DriverManager.getConnection(url, userid, passwd); 4
 }
 else {
 System.out.println("Usage: java DB2Appl [username password]");
 System.exit(0);
 }

 // retrieve data from the database
 System.out.println("Retrieve some data from the database...");
 Statement stmt = con.createStatement(); 5
 ResultSet rs = stmt.executeQuery("SELECT * from employee"); 6

 System.out.println("Received results:");

 // display the result set
 // rs.next() returns false when there are no more rows
 while (rs.next()) { 7
 String a = rs.getString(1); 8
 String str = rs.getString(2);

 System.out.print(" empno= " + a);
 System.out.print(" firstname= " + str);
 System.out.print("");
 }

 rs.close();
 }
 }
}
```

```

 stmt.close();

 // update the database
 System.out.println("Update the database... ");
 stmt = con.createStatement();
 int rowsUpdated = stmt.executeUpdate("UPDATE employee
SET firstnme = 'SHILI' where empno = '000010'"); 9

 System.out.print("Changed "+rowsUpdated);

 if (1 == rowsUpdated)
 System.out.println(" row.");
 else
 System.out.println(" rows.");

 stmt.close();
 con.close();
 } catch(Exception e) {
 System.out.println(e);
 }
}
}

```

## Distributing a JDBC Application

Distribute your JDBC application as you would any other Java application. As the application uses the DB2 client to communicate with the DB2 server, you have no special security concerns; authority verification is performed by the DB2 client.

To run your application on a client machine, you must install on that machine:

- A Java interpreter, which you need to run any Java code
- A DB2 client, which also includes the DB2 JDBC driver

To build your application, you must also install the JDK for your operating system. For information on setting up your Java environment, building DB2 Java applications, and running DB2 Java applications, refer to the *Application Building Guide*.

## Distributing and Running a JDBC Applet

Like other Java applets, you distribute your JDBC applet over the network (intranet or Internet). Typically you would embed the applet in a hypertext markup language (HTML) page. For example, to call the sample applet DB2Applet.java, (provided in sqllib/samples/java) you might use the following <APPLET> tag:

```
<applet code="DB2Appl.class" width=325 height=275 archive="db2java.zip">
 <param name="server" value="webhost">
 <param name="port" value="6789">
</applet>
```

To run your applet, you need only a Java-enabled Web browser on the client machine. When you load your HTML page, the applet tag instructs your browser to download the Java applet and the `db2java.zip` class library, which includes the DB2 JDBC driver implemented by the `COM.ibm.db2.jdbc.net` class. When your applet calls the JDBC API to connect to DB2, the JDBC driver establishes separate communications with the DB2 database through the JDBC applet server running on the Web server.

For information on building and distributing Java applets, refer to the *Application Building Guide*.

## JDBC 2.0

JDBC 2.0 is the latest version of JDBC from Sun. This version of JDBC has two defined parts: the **core API**, and the **standard extension API**. For information on the JDBC specification, see the DB2 Universal Database Java website at <http://www.software.ibm.com/data/db2/java/>.

The JDBC 2.0 driver is currently available only on Windows 32-bit operating systems. Before you can use the JDBC 2.0 driver, you must follow the instructions in “Performing the Installation” on page 545.

### Features

The DB2 JDBC 2.0 driver supports the JDBC 2.0 core API, however, it does not support all of the features defined in the specification. Here is a list of main features that the DB2 JDBC 2.0 driver supports:

- Scrollable ResultSet
- Batch Update (Batch statements and batch prepared statements)
- LOB support

Here is a list of features that the DB2 JDBC 2.0 driver does **NOT** support:

- Updatable Scrollable ResultSet
- New SQL types (Array, Ref, Distinct, Java Object)
- Customized SQL type mapping

### Compatibility

This version of the specification is backward compatible with the previous version (1.22). However, the DB2 JDBC 1.22 driver supports LOB types as an extension of the JDBC 1.22 specification, and this extension is not part of the

new specification's backward compatibility. This means that existing JDBC applications that rely on the LOB support of the JDBC 1.22 driver may not work with the new driver. For information on the DB2 JDBC 1.22 driver support for LOBs and graphic types, see "Using LOBs and Graphical Objects With JDBC 1.22" on page 566. To fix the problem, we recommend that you modify the application to comply with the new specification.

However, this solution may not be practical for every situation. As a workaround we provide a keyword, `JDBCVERSION`, that you can set to "122" to tell the JDBC 2.0 driver to use the 1.22 version of LOB support. The default is "200" to tell the JDBC 2.0 driver to use the 2.0 version of LOB support. You can set this keyword in `db2cli.ini`, or pass it in as a connection attribute in the `getConnection` property argument.

**Note:** If you use the JDBC 1.22 driver, the `JDBCVERSION` keyword does not affect LOB support for JDBC.

### Performing the Installation

As the JDBC 2.0 driver is not available for every platform, the JDBC 1.22 driver is still the default driver. To take advantage of the new features of JDBC 2.0, you must install both the JDBC 2.0 driver and JDK 1.2 support for your platform.

To install the JDBC 2.0 driver, enter the `usejdbc2` command from the `sqllib\java12` directory. This command performs the following tasks:

- Creates an `sqllib\java11` directory for the 1.22 driver files
- Backs up the JDBC 1.22 driver files into the `sqllib\java11` directory
- Copies the JDBC 2.0 driver files from the `sqllib\java12` directory into the appropriate directories

To switch back to the JDBC 1.22 driver, execute the `usejdbc1` batch file from the `sqllib\java12` directory.

---

## SQLJ Programming

DB2 SQLJ support is based on the SQLJ ANSI standard. Refer to the DB2 Java website at <http://www.software.ibm.com/data/db2/java> for a pointer to the ANSI website and other SQLJ resources. This chapter contains an overview of SQLJ programming and information that is specific to DB2 SQLJ support.

The following kinds of SQL constructs may appear in SQLJ programs:

- Queries; for example, `SELECT` statements and expressions.

- SQL Data Change Statements (DML); for example, INSERT, UPDATE, DELETE.
- Data Statements; for example, FETCH, SELECT..INTO.
- Transaction Control; for example, COMMIT, ROLLBACK, etc.
- Data Definition Language (DDL, also known as Schema Manipulation Language); for example, CREATE, DROP, ALTER.
- Calls to stored procedures; for example, CALL MYPROC(:x, :y, :z)
- Invocations of stored functions; for example, VALUES( MYFUN(:x) )

## DB2 SQLJ Support

DB2 SQLJ support is provided by the DB2 Software Developer's Kit (DB2 SDK). Along with the JDBC support provided by the DB2 client, DB2 SQLJ support allows you to create, build, and run embedded SQL for Java applications, applets, stored procedures and user-defined functions (UDFs). These contain static SQL and use embedded SQL statements that are bound to a DB2 database.

The SQLJ support provided by the DB2 SDK includes:

- The SQLJ translator, SQLJ, which replaces embedded SQL statements in the SQLJ program with Java source statements and generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the `sqllib/java/sqlj.zip` file.
- The SQLJ run-time classes, available in `sqllib/java/runtime.zip`.
- The DB2 SQLJ profile customizer, `db2profc`, which precompiles the SQL statements stored in the generated profile and generates a package in the DB2 database.
- The DB2 SQLJ profile printer, `db2profp`, which prints the contents of a DB2 customized profile in plain text.
- The SQLJ profile auditor installer, `profdb`, which installs (or uninstalls) debugging class-auditors into an existing set of binary profiles. Once installed, all `RTStatement` and `RTResultSet` calls made during application run time are logged to a file (or standard output), which can then be inspected to verify expected behavior and trace errors. Note that only those calls made to the underlying `RTStatement` and `RTResultSet` call interface at run time are audited.
- The SQLJ profile conversion tool, `profconv`, which converts a serialized profile instance to class bytecode format. Some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. As a work-around, you need to run this utility to perform the conversion.

For more information on the db2profc and db2profp commands, refer to the *Command Reference*. For more information on the SQLJ run-time classes, refer to the DB2 Java website at <http://www.software.ibm.com/data/db2/java>.

## DB2 SQLJ Restrictions

When you create DB2 applications with SQLJ, you should be aware of the following restrictions:

- DB2 SQLJ support adheres to standard DB2 Universal Database restrictions on issuing SQL statements.
- A positioned UPDATE and DELETE statement is not a valid sub-statement in a Compound SQL statement.
- The precompile option "DATETIME" is not supported. Only the date and time formats of the International Standards Organization are supported.
- The precompile option "PACKAGE USING package-name" specifies the name of the package that is to be generated by the translator. If a name is not entered, the name of the profile (minus extension and folded to uppercase) is used. Maximum length is 8 characters. Since the SQLJ profile name has the suffix \_SJProfileN, where N is the profile key number, the profile name will always be longer than 8 characters. The default package name will be constructed by concatenating the first (8 - *pfKeyNumLen*) characters of the profile number and the profile key number, where *pfKeyNumLen* is the length of the profile key number in the profile name. If the length of the profile key number is longer than 7, the last 7 digits will be used without any warnings. For example:

profile name	default package name
-----	-----
App_SJProfile1	App_SJP1
App_SJProfile123	App_S123
App_SJProfile1234567	A1234567
App_SJProfile12345678	A2345678

- When a `java.math.BigDecimal` host variable is used, the precision and scale of the host variable is not available during the translation of the application. If the precision and scale of the decimal host variable is not obvious from the context of the statement in which it is used, the precision and scale can be specified using a CAST.
- A Java variable with type `java.math.BigInteger` cannot be used as a host variable in an SQL statement.

Some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. You will get the following error message when trying to load the applet `Applt` in those browsers:

```
java.lang.ClassNotFoundException: Applt_SJProfile0
```

As a work-around, there is a utility which converts a serialized profile into a profile stored in Java class format. The utility is a Java class called `sqlj.runtime.profile.util.SerProfileToClass`. It takes a serialized profile resource file as input and produces a Java class containing the profile as output. Your profile can be converted using the following command:

```
profconv Applet_SJProfile0.ser
```

or

```
java sqlj.runtime.profile.util.SerProfileToClass Applet_SJProfile0.ser
```

The class `Applet_SJProfile0.class` is created as a result. Replace all profiles in `.ser` format used by the applet with profiles in `.class` format.

For an SQLJ applet, you need both `db2java.zip` and `runtime.zip` files. If you choose not to package all your applet classes, classes in `db2java.zip` and `runtime.zip` into a single Jar file, put both `db2java.zip` and `runtime.zip` (separated by a comma) into the archive parameter in the "applet" tag. For those browsers that do not support multiple zip files in the archive tag, specify `db2java.zip` in the archive tag, and unzip `runtime.zip` with your applet classes in a working directory that is accessible to your web browser.

## Embedding SQL Statements

Static SQL statements in SQLJ appear in *SQLJ clauses*. SQLJ clauses are the mechanism by which SQL statements in Java programs are communicated to the database.

The SQLJ translator recognizes SQLJ clauses and SQL statements because of their structure, as follows:

- SQLJ clauses begin with the token `#sql`
- SQLJ clauses end with a semicolon

The simplest SQLJ clauses are *executable clauses* and consist of the token `#sql` followed by an SQL statement enclosed in braces. For example, the following SQLJ clause may appear wherever a Java statement may legally appear. Its purpose is to delete all rows in the table named TAB:

```
#sql { DELETE FROM TAB };
```

In an SQLJ executable clause, the tokens that appear inside the braces are SQL tokens, except for the host variables. All host variables are distinguished by the colon character so the translator can identify them. SQL tokens never occur outside the braces of an SQLJ executable clause. For example, the following Java method inserts its arguments into an SQL table. The method body consists of an SQLJ executable clause containing the host variables `x`, `y`, and `z`:



```

void m (int x, String y, float z) throws SQLException
{
 #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}

```

In general, SQL tokens are case insensitive (except for identifiers delimited by double quotation marks), and can be written in upper, lower, or mixed case. Java tokens, however, are case sensitive. For clarity in examples, case insensitive SQL tokens are uppercase, and Java tokens are lowercase or mixed case. Throughout this chapter, the lowercase null is used to represent the Java "null" value, and the uppercase NULL to represent the SQL null value.

## Declaring Iterators for Positioned Updates and Deletes

Unlike SQL statements that retrieve data from a table, positioned UPDATE and DELETE operations require two Java source files. In one source file, you must declare the iterator. In the declaration, you must use an SQLJ implements clause to implement the `sqlj.runtime.ForUpdate` interface. You must also declare the iterator as public. For example, suppose that you declare iterator `DelByName` like this in `file1.sqlj`:

```
#sql public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo);
```

You can then use the translated and compiled iterator in a different source file. To use the iterator:

1. Declare an instance of the generated iterator class
2. Assign the SELECT statement for the positioned UPDATE or DELETE to the iterator instance
3. Execute positioned UPDATE or DELETE statements using the iterator

To use `DelByName` for a positioned DELETE in `file2.sqlj`, execute statements like those in "Deleting Rows Using a Positioned Iterator" .

```

{
 DelByName deliter; // Declare object of DelByName class
 String enum;
1 #sql deliter = { SELECT EMPNO FROM EMP WHERE WORKDEPT='D11'};
 while (deliter.next())
 {
2 enum = deliter.EmpNo(); // Get value from result table
3 #sql { DELETE WHERE CURRENT OF :deliter };
 // Delete row where cursor is positioned
 }
}

```

## Notes:

1. **1** This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable *deliter*.
2. **2** This statement positions the iterator to the next row to be deleted.
3. **3** This SQLJ clause performs the positioned DELETE.

## SQLJ Example: App.sqlj

The following example SQLJ application, App.sqlj, uses static SQL to retrieve and update data from the EMPLOYEE table of the DB2 sample database.

1. **Declare iterators.** This section declares two types of iterators:

### App\_Cursor1

Declares column data types and names, and returns the values of the columns according to column name (Named binding to columns).

### App\_Cursor2

Declares column data types, and returns the values of the columns by column position (Positional binding to columns).

2. **Initialize the iterator.** The iterator object *cursor1* is initialized using the result of a query. The query stores the result in *cursor1*.
3. **Advance the iterator to the next row.** The *cursor1.next()* method returns a Boolean false if there are no more rows to retrieve.
4. **Move the data.** The named accessor method *empno()* returns the value of the column named *empno* on the current row. The named accessor method *firstnme()* returns the value of the column named *firstnme* on the current row.
5. **SELECT data into a host variable.** The SELECT statement passes the number of rows in the table into the host variable *count1*.
6. **Initialize the iterator.** The iterator object *cursor2* is initialized using the result of a query. The query stores the result in *cursor2*.
7. **Retrieve the data.** The FETCH statement returns the current value of the first column declared in the ByPos cursor from the result table into the host variable *str2*.
8. **Check the success of a FETCH..INTO statement.** The *endFetch()* method returns a Boolean true if the iterator is not positioned on a row, that is, if the last attempt to fetch a row failed. The *endFetch()* method returns false if the last attempt to fetch a row was successful. DB2 attempts to fetch a row when the *next()* method is called. A FETCH...INTO statement implicitly calls the *next()* method.

9. **Close the iterators.** The `close()` method releases any resources held by the iterators. You should explicitly close iterators to ensure that system resources are released in a timely fashion.

### JDBC Example: App.sqlj:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; 1
#sql iterator App_Cursor2 (String) ;

class App
{
 /*****
 ** Register Driver **
 *****/

 static
 {
 try
 {
 Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 }

 /*****
 ** Main **
 *****/

 public static void main(String argv[])
 {
 try
 {
 App_Cursor1 cursor1;
 App_Cursor2 cursor2;

 String str1 = null;
 String str2 = null;
 long count1;

 // URL is jdbc:db2:dbname
 String url = "jdbc:db2:sample";

 DefaultContext ctx = DefaultContext.getDefaultContext();
 if (ctx == null)
 {
 try
 {
 // connect with default id/password
 Connection con = DriverManager.getConnection(url);
 con.setAutoCommit(false);
 ctx = new DefaultContext(con);
 }
 }
 }
 }
}
```

```

 catch (SQLException e)
 {
 System.out.println("Error: could not get a default context");
 System.err.println(e);
 System.exit(1);
 }
 DefaultContext.setDefaultContext(ctx);
 }

 // retrieve data from the database
 System.out.println("Retrieve some data from the database.");
 #sql cursor1 = {SELECT empno, firstnme FROM employee}; 2

 // display the result set
 // cursor1.next() returns false when there are no more rows
 System.out.println("Received results:");
 while (cursor1.next()) 3
 {
 str1 = cursor1.empno(); 4
 str2 = cursor1.firstnme();

 System.out.print (" empno= " + str1);
 System.out.print (" firstnme= " + str2);
 System.out.print ("");
 }
 cursor1.close(); 9

 // retrieve number of employee from the database
 #sql { SELECT count(*) into :count1 FROM employee }; 5
 if (1 == count1)
 System.out.println ("There is 1 row in employee table");
 else
 System.out.println ("There are " + count1
 + " rows in employee table");

 // update the database
 System.out.println("Update the database. ");
 #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

 // retrieve the updated data from the database
 System.out.println("Retrieve the updated data from the database.");
 str1 = "000010";
 #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; 6

 // display the result set
 // cursor2.next() returns false when there are no more rows
 System.out.println("Received results:");
 while (true)
 {
 #sql { FETCH :cursor2 INTO :str2 }; 7
 if (cursor2.endFetch()) break; 8

 System.out.print (" empno= " + str1);
 System.out.print (" firstnme= " + str2);
 System.out.print ("");
 }

```

```

 }
 cursor2.close(); 9

 // rollback the update
 System.out.println("Rollback the update.");
 #sql { ROLLBACK work };
 System.out.println("Rollback done.");
}
catch(Exception e)
{
 e.printStackTrace();
}
}
}

```

## Host Variables

Arguments to embedded SQL statements are passed through *host variables*, which are variables of the host language that appear in the SQL statement. Host variables have up to three parts:

- A colon prefix, `:`.
- An optional parameter mode identifier: IN, OUT, or INOUT.
- A Java host variable that is a Java identifier for a parameter, variable, or field.

The evaluation of a Java identifier does not have side effects in a Java program, so it may appear multiple times in the Java code generated to replace an SQLJ clause.

The following query contains the host variable, `:x`, which is the Java variable, field, or parameter `x` visible in the scope containing the query:

```
SELECT COL1, COL2 FROM TABLE1 WHERE :x > COL3
```

All host variables specified in compound SQL are input host variables by default. You have to specify the parameter mode identifier OUT or INOUT before the host variable in order to mark it as an output host variable. For example:

```
#sql {begin compound atomic static
 select count(*) into :OUT count1 from employee;
 end compound}

```

## Calls to Stored Procedures and Functions

Databases may contain *stored procedures* and *stored functions*. User-defined procedures and functions are named schema objects that execute in the database. An SQLJ executable clause appearing as a Java statement may call a stored procedure by means of a CALL statement like the following:

```
#sql { CALL SOME_PROC(:INOUT myarg) };
```

Stored procedures may have IN, OUT, or INOUT parameters. In the above case, the value of host variable *myarg* is changed by the execution of that clause. An SQLJ executable clause may call a stored function by means of the SQL VALUES construct. For example, assume a stored function F that returns an integer. The following example illustrates a call to that function that then assigns its result to Java local variable *x*:

```
{
 int x;
 #sql x = { VALUES(F(34)) };
}
```

## Compiling and Running SQLJ Programs

To run an SQLJ program with program name *MyClass*, do the following:

1. Translate the Java source code with Embedded SQL to generate the Java source code *MyClass.java* and profiles *MyClass\_SJProfile0.ser*, *MyClass\_SJProfile1.ser*, ... (one profile for each connection context):

```
sqlj MyClass.sqlj
```

When you use the SQLJ translator without specifying an *sqlj.properties* file, the translator uses the following values:

```
sqlj.url=jdbc:db2:sample
sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
sqlj.online=sqlj.semantics.JdbcChecker
sqlj.offline=sqlj.semantics.OfflineChecker
```

If you do specify an *sqlj.properties* file, make sure the following options are set:

```
sqlj.url=jdbc:db2:dbname
sqlj.driver=COM.ibm.db2.jdbc.app.DB2Driver
sqlj.online=sqlj.semantics.JdbcChecker
sqlj.offline=sqlj.semantics.OfflineChecker
```

where *dbname* is the name of the database. You can also specify these options on the command line. For example, to specify the database *mydata* when translating *MyClass*, you can issue the following command:

```
sqlj -url=jdbc:db2:mydata MyClass.sqlj
```

Note that the SQLJ translator automatically compiles the translated source code into class files, unless you explicitly turn off the compile option with the *-compile=false* clause.

2. Install DB2 SQLJ Customizers on generated profiles and create the DB2 packages in the DB2 database *dbname*:

```
db2profcc -user=user-name -password=user-password -url=jdbc:db2:dbname
-preoptions="bindfile using MyClass0.bnd package using MyClass0"
MyClass_SJProfile0.ser
```

```
db2profrc -user=user-name -password=user-password -url=jdbc:db2:dbname
-preoptions="bindfile using MyClass1.bnd package using MyClass1"
MyClass_SJProfile1.ser
...
```

### 3. Execute the SQLJ program:

```
java MyClass
```

The translator generates the SQL syntax for the database for which the SQLJ profile is customized. For example,

```
i = { VALUES (F(:x)) };
```

is translated by the SQLJ translator and stored as

```
? = VALUES (F (?))
```

in the generated profile. When connecting to a DB2 Universal Database database, DB2 will customize the VALUE statement into:

```
VALUES(F(?)) INTO ?
```

but when connecting to a DB2 Universal Database for OS/390 database, DB2 customizes the VALUE statement into:

```
SELECT F(?) INTO ? FROM SYSIBM.SYSDUMMY1
```

If you run the DB2 SQLJ profile customizer, db2profrc, against a DB2 Universal Database database and generate a bindfile, you cannot use that bindfile to bind up to a DB2 for OS/390 database when there is a VALUES clause in the bindfile. This also applies to generating a bindfile against a DB2 for OS/390 database and trying to bind with it to a DB2 Universal Database database.

For detailed information on building and running DB2 SQLJ programs, refer to the *Application Building Guide*.

## SQLJ Precompile Options

The following translator options are not applicable:

```
CONNECT
DISCONNECT
DYNAMICBIND
NOLINEMACRO
OPTLEVEL
OUTPUT
SQLCA
SQLFLAG
SQLRULES
SYNCPPOINT
TARGET
WCHARTYPE
```



To print the content of the profiles generated by the SQLJ translator in plain text, use the `profp` utility as follows:

```
profp MyClass_SJProfile0.ser
profp MyClass_SJProfile1.ser
...
```

To print the content of the DB2 customized version of the profile in plain text, use the `db2profp` utility as follows, where *dbname* is the name of the database:

```
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
MyClass_SJProfile0.ser
db2profp -user=user-name -password=user-password -url=jdbc:db2:dbname
MyClass_SJProfile1.ser
...
```

---

## Stored Procedures and UDFs

You can create and use stored procedures and UDFs in Java just like you can for other programming languages. There are some programming considerations (as discussed in “Function Definitions” on page 564) that you need to know when you write your Java code. You also need to *register* your stored procedure and UDFs. For information on how to register your stored procedure, see “Chapter 6. Stored Procedures” on page 185. For information on how to register your UDF, refer to the `CREATE FUNCTION` statement in the *SQL Reference*.

To run your UDFs and stored procedures on the server, DB2 calls the Java interpreter. DB2 does not include a Java interpreter; your database administrator must install and configure the appropriate Java Development Kit (JDK) on your DB2 server before starting up the database.

The runtime libraries for the Java interpreter must be available in the system search paths (`PATH` or `LIBPATH` or `LD_LIBRARY_PATH`, and `CLASSPATH`). For more information on setting up the Java environment, refer to the *Application Building Guide*.

DB2 loads or starts the Java interpreter on the first call to a Java UDF or stored procedure. For NOT FENCED UDFs and stored procedures, DB2 loads one Java interpreter per database instance, and runs it inside the database engine’s address space for best performance. For FENCED UDFs, DB2 uses a distinct interpreter inside the `db2udf` process; similarly, FENCED stored procedures use a distinct interpreter inside the `db2dar` process. In all cases, the Java interpreter stays loaded until the embedding process ends.

**Note:** If you are running a *database server with local clients* node type, you must set the `maxdari` database manager configuration parameter to a non-zero value before you invoke a Java stored procedure.

You can study the Java stored procedure samples that are provided in the `sqllib/samples/java` directory. For a list of the sample programs included with DB2, see “Appendix B. Sample Programs” on page 633.

Remember that all Java class files that you use to implement a stored procedure or UDF must reside in either a JAR file you have installed in the database, or in the correct stored procedure or UDF path for your operating system as discussed in “Where to Put Java Classes”.

**Note:** On a mixed code page database server, Java user-defined functions and stored procedures cannot use CLOB type arguments, because random access on character boundaries on large mixed code page strings has not yet been implemented. Full support for all LOB types is intended for SBCS databases. For mixed databases, support is intended for the BLOB and the DBCLOB types. As a workaround, applications running on a mixed database system should convert CLOB arguments to DBCLOB, LONG VARGRAPHIC, or LONG VARCHAR types. For UDFs, this can be done with the CAST operator.

## Where to Put Java Classes

You can choose to use individual Java class files for your stored procedures and UDFs, or collect the class files into JAR files and install the JAR file in the database. If you decide to use JAR files, refer to “Java Stored Procedures and UDFs” on page 562 for further instructions.

To enable DB2 to find and use your stored procedures and UDFs, you must store the corresponding class files in the *function directory*, which is a directory defined for your operating system as follows:

### Unix operating systems

`sqllib/function`

### OS/2 or Windows 32-bit operating systems

`instance_name\function`, where *instance\_name* represents the value of the `DB2INSTPROF` instance-specific registry setting. If DB2 does not find the stored procedure in `instance_name\function`, DB2 searches the directories defined by the `PATH` and `LIBPATH` environment variables.

For example, the function directory for a Windows NT server with DB2 installed in the `C:\sqllib` directory, and with no specified `DB2INSTPROF` registry setting, is:

C:\sqllib\function

If you choose to use individual class files, you must store the class files in the appropriate directory for your operating system. If you declare a class to be part of a Java package, create the corresponding subdirectories in the function directory and place the files in the corresponding subdirectory. For example, if you create a class `ibm.tests.test1` for a Linux system, store the corresponding Java bytecode file (named `test1.class`) in `sqllib/function/ibm/tests`.

The Java interpreter that DB2 invokes uses the `CLASSPATH` environment variable to locate Java files. DB2 adds the function directory and `sqllib/java/db2java.zip` to the front of your `CLASSPATH` setting.

To set your environment so that the Java interpreter can find where you have stored the Java class files, you may need to set the `jdk11_path` configuration parameter, or else use the default value. Also, you may need to set the `java_heap_sz` configuration parameter to increase the heap size for your application. For more information on configuration parameters, refer to the *Administration Guide, Design and Implementation*.

## Debugging Stored Procedures

DB2 provides the capability to interactively debug a stored procedure written in JDBC when the stored procedure executes on an AIX or Windows NT server. Debugging support requires JDK level 1.1.6 or higher. The easiest way to invoke debugging is through the DB2 Stored Procedure Builder. See the online help for the Stored Procedure Builder for more information about how to do this.

This section includes the following topics:

- Preparing to debug
- Populating the debug table
- Invoking the debugger

### Preparing to Debug

1. Compile the stored procedure in debug mode according to your Java development environment documentation.
2. **Prepare the server.**
  - If the source code is stored on the server, set the `CLASSPATH` environment variable to include the Java source code directory or store the source code in the function directory, as defined in “Where to Put Java Classes” on page 558.
  - Use the `db2set` command to enable debugging for your instance:

```
db2set DB2ROUTINE_DEBUG=ON
```

3. Set the client environment variables.

- If the source code is stored on the client, set the DB2\_DBG\_PATH environment variable to the directory which contains the source code for the stored procedure.

4. Create the debug table.

If you do not use the Stored Procedure Builder to invoke the debugger, create the debug table with the following command:

```
db2 -tf sqllib/misc/db2debug.ddl
```

**Note:** On DB2 Enterprise - Extended Edition systems, the default nodegroup is IBMDEFAULTGROUP for the USERSPACE1 table space, and consists of all nodes defined for the system. To improve the performance of debugging stored procedures in a DB2 Enterprise - Extended Edition configuration, you should have a single coordinator node where debugging will occur and define a nodegroup that only contains that node.

5. Start the debugger daemon on the client.

From the stored procedure client, start the debugger daemon with the following command:

```
db2dbugd -qport=portno
```

where *portno* is an unused TCP/IP port number. If you do not supply a value, the debugger uses 8000 as the default port number. On Windows 32-bit operating systems, you can also click the debugger daemon shortcut located in the DB2 folder to start the debugger with the default port number.

## Populating the Debug Table

The debug table contains information about the stored procedures you debug and the client/server environment that you debug in. Only DBAs or users with INSERT, UPDATE, or DELETE privilege can manipulate values directly in the base table DB2DBG.ROUTINE\_DEBUG. However, unless the DBA has added further restrictions, anyone can add, update, or delete rows through the user view, DB2DBG.ROUTINE\_DEBUG\_USER. Therefore, the rest of this section assumes that you are populating that table through the user view.

If you use the Stored Procedure Builder to invoke debugging, you can use the debugger utility to populate and manage the debug table. Otherwise, to enable debugging support for a given stored procedure, issue the following command from the CLP:

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('authid', 'S', 'schema', 'proc_name', 'Y', 'IP_num')
```

where:

**authid** The user name used for debugging the stored procedure, that is, the user name used to connect to the database.

**schema**  
The schema name for the stored procedure.

**proc\_name**  
The specific name of the stored procedure. This is the specific name that was provided on the CREATE PROCEDURE command or a system generated identifier, if no specific name has been provided.

**IP\_num**  
The IP address in the form nnn.nnn.nnn.nnn of the client used to debug the stored procedure.

For example, to enable debugging for the stored procedure *MySchema.myProc* by the user *USER1* with the debugging client located at the IP address 123.234.111.222, type the following command:

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('USER1', 'S', 'MySchema', 'myProc', 'Y', '123.234.111.222')
```

If you drop a stored procedure, its debug information is not automatically deleted from the debug table. Debug information for non-existent stored procedures cannot harm your database or instance. If you want to keep the debug table synchronized with the DB2 catalog, you must delete the debug information manually.

Whether you create the debug table manually or through the Stored Procedure Builder, the debug table is named DB2DBG.ROUTINE\_DEBUG and has the following definition:

Table 21. DB2DBG.ROUTINE\_DEBUG Table Definition

Column Name	Data Type	Attributes	Description
AUTHID	VARCHAR(128)	NOT NULL, DEFAULT USER	The application authid under which the debugging for this stored procedure is to be performed. This is the userid that was provided on connect to the database.
TYPE	CHAR(1)	NOT NULL	Valid values: 'S' (Stored Procedure)
ROUTINE_SCHEMA	VARCHAR(128)	NOT NULL	Schema name of the stored procedure to be debugged
SPECIFICNAME	VARCHAR(18)	NOT NULL	Specific name of the stored procedure to be debugged

Table 21. DB2DBG.ROUTINE\_DEBUG Table Definition (continued)

Column Name	Data Type	Attributes	Description
DEBUG_ON	CHAR(1)	NOT NULL, DEFAULT 'N'	Valid values: <ul style="list-style-type: none"> <li>• <b>Y</b> - enables debugging for the stored procedure named in ROUTINE_SCHEMA.SPECIFICNAME</li> <li>• <b>N</b> - disables debugging for stored procedure named in ROUTINE_SCHEMA.SPECIFICNAME. This is the default.</li> </ul>
CLIENT_IPADDR	VARCHAR(15)	NOT NULL	The IP address of the client that does the debugging of the form nnn.nnn.nnn.nnn
CLIENT_PORT	INTEGER	NOT NULL, DEFAULT 8000	The port of the debugging communication. The default is 8000.
DEBUG_STARTN	INTEGER	NOT NULL	Not used.
DEBUG_STOPN	INTEGER	NOT NULL	Not used.
The primary key of this table is AUTHID, TYPE, ROUTINE_SCHEMA, SPECIFICNAME.			

### Invoking the Debugger

If you have successfully followed the previous steps, calling a stored procedure invokes the debugger on the client with the IP address that you specified in the debug table.

In the debugger, you can step through the source code, display variables, and set breakpoints in the source code. For detailed information on using the debugger, see the debugger documentation contained in the online help.

### Java Stored Procedures and UDFs

Java stored procedures and UDFs, hereafter referred to as *Java functions*, must be registered in the DB2 catalog. DB2 Universal Database Version 6 supports the *SQLJ Routines* core specification for registering and deploying Java functions. Use PARAMETER STYLE JAVA in your CREATE PROCEDURE and CREATE FUNCTION statements to specify compliance with SQLJ Routines.

Alternatively, DB2 supports DB2 V5 and V5.2 PARAMETER STYLE DB2GENERAL stored procedures and UDFs. For more information, see “Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs” on page 655.

To register a Java function or stored procedure, follow these steps:

1. Create the Java function as a Java method. Compile the Java source code into a Java class file. For information on creating Java stored procedures,

see “Chapter 6. Stored Procedures” on page 185. For information on creating Java UDFs, see “Creating and Using Java User-Defined Functions” on page 312.

2. Collect the class file containing the Java function in a *jar* file. You can collect one or more class files in a single JAR file. For instructions on creating JAR files, refer to the *Application Building Guide*.
3. Install the JAR file in the DB2 instance. For instructions on how to use the CALL SQLJ.INSTALL\_JAR command from the command line, see “Installing, Replacing, and Removing JAR Files”. You can also CALL the sqlj.install\_jar procedure in an application.
4. Issue the appropriate CREATE PROCEDURE or CREATE FUNCTION SQL statement for the Java function.
  - For a description and examples of using the CREATE PROCEDURE statement, see “Registering Stored Procedures” on page 190.
  - For a description and examples of using the CREATE FUNCTION statement, refer to the *SQL Reference*.

When you install a JAR file, DB2 extracts the Java class files from the JAR file and registers each procedure in the system catalog. DB2 copies the JAR file to a *jar/schema* subdirectory of the function directory. DB2 gives the new copy of the JAR file the name given in the *jar-id* clause. Do not directly modify a JAR file which has been installed in the DB2 instance. Instead, you can use the CALL SQLJ.REMOVE\_JAR and CALL SQLJ.REPLACE\_JAR commands to remove or replace an installed JAR file.

### Installing, Replacing, and Removing JAR Files

To install or replace a JAR file in the DB2 instance, you can use the following command syntax at the Command Line Processor:

```

>> CALL SQLJ.INSTALL_JAR (— ' — jar-url ————— ' — , — ' — jar-id ————— ' —) —————>>
 |
 | SQLJ.REPLACE_JAR

```

#### Notes:

1. Specifies the URL containing the JAR file to be installed or replaced. The only URL scheme supported is 'file:'.
2. Specifies the JAR identifier in the database to be associated with the file specified by the jar-url.

**Note:** On OS/2 and Windows 32-bit operating systems, DB2 stores JAR files in the path specified by the *DB2INSTPROF* instance-specific registry

setting. To make JAR files unique for an instance, you must specify a unique value for *DB2INSTPROF* for that instance.

For example, to install the *Proc.jar* file located in the *file:/home/db2inst/classes/* directory in the DB2 instance, issue the following command from the Command Line Processor:

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

Subsequent SQL commands that use of the *Procedure.jar* file refer to it with the name *myproc\_jar*. To remove a JAR file from the database, use the *CALL REMOVE\_JAR* command with the following syntax:

```
➤➤CALL SQLJ.REMOVE_JAR(——(——'——jar-id——'——)——)➤➤
 (1)
```

#### Notes:

1. Specifies the JAR identifier of the JAR file that is to be removed from the database

To remove the JAR file *myProc\_jar* from the database, enter the following command at the Command Line Processor:

```
CALL SQLJ.REMOVE_JAR('myProc_jar')
```

#### Function Definitions

To create a Java function, you must code the corresponding public static method in a public class. A Java function must also be declared with the throws *SQLException* clause. Code the method signature and the rest of the method declaration to correspond with the output expected from the method body.

**Functions That Return No Values:** To create a method that returns no values to the calling program, declare the method to return *void* and include any parameters in the signature that need to be passed to the method body. You can write a stored procedure that performs a simple *UPDATE* and returns no value to the client application as follows:

```
public class JavaExamples {
 public static void updateJob(String oldJob, String newJob)
 throws SQLException {
 Connection conn=DriverManager.getConnection("jdbc:ibm.db2.sample");
 PreparedStatement stmt = conn.prepareStatement("UPDATE employee
 SET job = ? WHERE job = ?");
 stmt.setString(1, newJob);
 stmt.setString(2, oldJob);
 stmt.executeUpdate();
 }
}
```



```

 conn.close();
 return;
 }
}

```

**Functions That Return A Single Value:** Declare Java methods that return a single value with the Java return type that corresponds to the respective SQL data type (see “Supported SQL Data Types” on page 535). You can write a scalar UDF that returns an SQL INTEGER value as follows:

```

public class JavaExamples {
 public static int getDivision(String division) throws SQLException {
 if (division.equals("Corporate")) return 1;
 else if (division.equals("Eastern")) return 2;
 else if (division.equals("Midwest")) return 3;
 else if (division.equals("Western")) return 4;
 else return 5;
 }
}

```

**Functions That Return Multiple Values:** Java methods which are cataloged as stored procedures may return one or more values. You can also write Java stored procedures that return multiple result sets; see “Returning Result Sets from Stored Procedures” on page 212. To code a method which will return a predetermined number of values, declare the return type void and include the types of the expected output as arrays in the method signature. You can write a stored procedure which returns the names, years of service, and salaries of the two most senior employees with a salary under a given threshold as follows:

```

public Class JavaExamples {
 public static void lowSenioritySalary
 (String[] name1, int[] years1, BigDecimal[] salary1,
 String[] name2, int[] years2, BigDecimal[] salary2,
 Integer threshold) throws SQLException {
 #sql iterator ByNames (String name, int years, BigDecimal salary);
 ByNames result;
 #sql result = {"SELECT name, years, salary
 FROM staff
 WHERE salary < :threshold
 ORDER BY years DESC"};
 if (result.next()) {
 name1[0] = result.name();
 years1[0] = result.years();
 salary1[0] = result.salary();
 }
 else {
 name1[0] = "****";
 return;
 }
 if (result.next()) {
 name2[0] = result.name();
 years2[0] = result.years();
 salary2[0] = result.salary();
 }
 }
}

```

```

 }
 else {
 name2[0] = "****";
 return;
 }
}
}

```

---

## Using LOBs and Graphical Objects With JDBC 1.22

The JDBC 2.0 specification for JDK 1.2 defines support for LOBs and graphic types. For more information on the DB2 JDBC 2.0 driver support, see “JDBC 2.0” on page 544.

However, the JDBC 1.22 specification does not explicitly mention large objects (LOBs) or graphic types. DB2 provides the following support for LOBs and graphic types if you use the JDBC 1.22 driver.

If you use LOBs or graphic types in your applications, treat LOBs as the corresponding LONGVAR type. Because LOB types are declared in SQL with a maximum length, ensure that you do not return arrays or strings longer than the declared limit. This consideration applies to SQL string types as well.

For GRAPHIC and DBCLOB types, treat them as the corresponding CHAR types. For conversions involving Unicode, the DB2 client first converts the data from the database format to the client format; DB2 then converts the data to Unicode. The following JDBC APIs involve conversion to or from Unicode:

### **getString**

Converts from database format to Unicode.

### **setString**

Converts from Unicode to database format.

### **getUnicodeStream**

Converts from database format to Unicode.

### **setUnicodeStream**

Converts from Unicode to database format.

The following JDBC APIs involve conversion to or from the local code page format:

### **setAsciiStream**

Converts from local code page to database format.

### **getAsciiStream**

Converts from database format to local code page format.

The SQLJ language provides direct support for static SQL operations that are known at the time the program is written. If some or all of a particular SQL statement cannot be determined until run time, it is a dynamic operation. To perform dynamic SQL operations from an SQLJ program, use JDBC. A `ConnectionContext` object contains a JDBC `Connection` object which can be used to create JDBC `Statement` objects needed for dynamic SQL operations.

Every SQLJ `ConnectionContext` class includes a constructor that takes as an argument a JDBC `Connection`. This constructor is used to create an SQLJ connection context instance that shares its underlying database connection with that of the JDBC connection.

Every SQLJ `ConnectionContext` instance has a `getConnection()` method that returns a JDBC `Connection` instance. The JDBC `Connection` returned shares the underlying database connection with the SQLJ connection context. It may be used to perform dynamic SQL operations as described in the JDBC API.

### Session Sharing

The interoperability methods described above provide a conversion between the connection abstractions used in SQLJ and those used in JDBC. Both abstractions share the same database session, that is, the underlying database connection. Accordingly, calls to methods that affect session state on one object will also be reflected in the other object, as it is actually the underlying shared session that is being affected.

JDBC defines the default values for session state of newly created connections. In most cases, SQLJ adopts these default values. However, whereas a newly created JDBC connection has auto commit mode on by default, an SQLJ connection context requires the auto commit mode to be specified explicitly upon construction.

### Connection Resource Management

Calling the `close()` method of a connection context instance causes the associated JDBC connection instance and the underlying database connection to be closed. Since connection contexts may share the underlying database connection with other connection contexts and/or JDBC connections, it may not be desirable to close the underlying database connection when a connection context is closed. A programmer may wish to release the resources maintained by the connection context (for example, statement handles) without actually closing the underlying database connection. To this end, connection context classes also support a `close()` method which takes a Boolean argument indicating whether or not to close the underlying database

connection: the constant `CLOSE_CONNECTION` if the database connection should be closed, and `KEEP_CONNECTION` if it should be retained. The variant of `close()` that takes no arguments is a shorthand for calling `close(CLOSE_CONNECTION)`.

If a connection context instance is not explicitly closed before it is garbage collected, then `close(KEEP_CONNECTION)` is called by the `finalize` method of the connection context. This allows connection related resources to be reclaimed by the normal garbage collection process while maintaining the underlying database connection for other JDBC and SQLJ objects that may be using it. Note that if no other JDBC or SQLJ objects are using the connection, then the database connection is closed and reclaimed by the garbage collection process.

Both SQLJ connection context objects and JDBC connection objects respond to the `close()` method. When writing an SQLJ program, it is sufficient to call the `close()` method on only the connection context object. This is because closing the connection context also closes the JDBC connection associated with it. However, it is not sufficient to close only the JDBC connection returned by the `getConnection()` method of a connection context. This is because the `close()` method of a JDBC connection does not cause the containing connection context to be closed, and therefore resources maintained by the connection context are not released until it is garbage collected.

The `isClosed()` method of a connection context returns `true` if any variant of the `close()` method has been called on the connection context instance. If `isClosed()` returns `true`, then calling `close()` has no effect, and calling any other method is undefined.

---

## Chapter 18. Programming in COBOL

Programming Considerations . . . . .	569	Host Structure Support. . . . .	582
Language Restrictions . . . . .	569	Indicator Tables . . . . .	584
Input and Output Files. . . . .	569	Using REDEFINES in COBOL Group	
Include Files . . . . .	570	Data Items . . . . .	585
Embedding SQL Statements . . . . .	573	Using BINARY/COMP-4 COBOL Data	
Host Variables. . . . .	575	Types . . . . .	585
Naming Host Variables. . . . .	575	Supported SQL Data Types . . . . .	586
Declaring Host Variables . . . . .	575	FOR BIT DATA . . . . .	589
Indicator Variables . . . . .	579	SQLSTATE and SQLCODE Variables . . . .	589
LOB Declarations in COBOL . . . . .	579	Japanese or Traditional Chinese EUC, and	
LOB Locator Declarations in COBOL	581	UCS-2 Considerations . . . . .	590
File Reference Declarations in COBOL	581	Object Oriented COBOL . . . . .	590

---

### Programming Considerations

Special host-language programming considerations are discussed in the following pages. Included is information on language restrictions, host language specific include files, embedding SQL statements, host variables, and supported data types for host variables.

---

### Language Restrictions

All API pointers are 4 bytes long. All integer variables used as value parameters in API calls must be declared with a USAGE COMP-5 clause.

---

### Input and Output Files

By default, the input file has an extension of .sqb, but if you use the TARGET precompile option (TARGET ANSI\_COBOL, TARGET IBMCOB, TARGET MFCOB or TARGET MFCOB16), the input file can have any extension you prefer.

By default, the output file has an extension of .cbl, but you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

---

## Include Files

The host-language-specific include files for COBOL have the file extension .cbl. If you use the "System/390 host data type support" feature of IBM COBOL compiler, the DB2 include files for your applications are in the following directory:

`$HOME/sql1lib/include/cobol_i`

If you build the DB2 sample programs with the supplied script files, you must change the include file path specified in the script files to the `cobol_i` directory and not the `cobol_a` directory.

If you do **not** use the "System/390 host data type support" feature of the IBM COBOL compiler, or you use an earlier version of this compiler, then the DB2 include files for your applications are in the following directory:

`$HOME/sql1lib/include/cobol_a`

The include files that are intended to be used in your applications are described below.

**SQL (sql.cbl)** This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

**SQLAPREP (sqlaprep.cbl)**

This file contains definitions required to write your own precompiler.

**SQLCA (sqlca.cbl)**

This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

**SQLCA\_92 (sqlca\_92.cbl)**

This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of the `sqlca.cbl` file when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.cbl` file is automatically included by the DB2 precompiler when the `LANGLEVEL` precompiler option is set to `SQL92E`.

**SQLCODES (sqlcodes.cbl)**

This file defines constants for the `SQLCODE` field of the SQLCA structure.

**SQLDA (sqlda.cbl)**

This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

**SQLEAU (sqleau.cbl)**

This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

**SQLENV (sqlenv.cbl)**

This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

**SQLETSO (sqltsd.cbl)**

This file defines the Table Space Descriptor structure, SQLETSDESC, which is passed to the Create Database API, sqlgcrea.

**SQLE819A (sqle819a.cbl)**

If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE819B (sqle819b.cbl)**

If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850A (sqle850a.cbl)**

If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850B (sqle850b.cbl)**

If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQL932A (sqle932a.cbl)**

If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL932B (sqle932b.cbl)**

If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252A (sql1252a.cbl)**

If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252B (sql1252b.cbl)**

If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLMON (sqlmon.cbl)**

This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

**SQLMONCT (sqlmonct.cbl)**

This file contains constant definitions and local data structure definitions required to call the Database System Monitor APIs.

**SQLSTATE (sqlstate.cbl)**

This file defines constants for the SQLSTATE field of the SQLCA structure.

**SQLUTBCQ (sqlutbcq.cbl)**

This file defines the Table Space Container Query data structure, SQLB-TBSCONTQRY-DATA, which is used with the table space container query APIs, sqlgstsc, sqlgftcq and sqlgtcq.

**SQLUTBSQ (sqlutbsq.cbl)**

This file defines the Table Space Query data structure, SQLB-TBSQRY-DATA, which is used with the table space query APIs, sqlgtsq, sqlgftsq and sqlgtsq.



## SQLUTIL (sqlutil.cbl)

This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

---

## Embedding SQL Statements

Embedded SQL statements consist of the following three elements:

Element	Correct COBOL Syntax
Keyword pair	EXEC SQL
Statement string	Any valid SQL statement
Statement terminator	END-EXEC.

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

The following rules apply to embedded SQL statements:

- Executable SQL statements must be placed in the PROCEDURE DIVISION. The SQL statements can be preceded by a paragraph name just as a COBOL statement.
- SQL statements can begin in either Area A (columns 8 through 11) or Area B (columns 12 through 72).
- Start each SQL statement with EXEC SQL and end it with END-EXEC. The SQL precompiler includes each SQL statement as a comment in the modified source file.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (--) followed by a string of zero or more characters and terminated by a line end. Do not place SQL comments after the SQL statement terminator as they will cause compilation errors because they would appear to be part of the COBOL language.
- COBOL comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
- SQL statements follow the same line continuation rules as the COBOL language. However, do not split the EXEC SQL keyword pair between lines.

- Do not use the COBOL COPY statement to include files containing SQL statements. SQL statements are precompiled before the module is compiled. The precompiler will ignore the COBOL COPY statement. Instead, use the SQL INCLUDE statement to include these files.

To locate the INCLUDE file, the DB2 COBOL precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- EXEC SQL INCLUDE payroll END-EXEC.

If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the precompiler searches for payroll.sqb, then payroll.cpy, then payroll.cbl, in each directory in which it looks.

- EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.

If the file name is enclosed in quotation marks, as above, no extension is added to the name.

If the file name in quotation marks does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for AIX, if DB2INCLUDE is set to '/disk2:myfiles/cobol', the precompiler searches for './pay/payroll.cbl', then '/disk2/pay/payroll.cbl', and finally './myfiles/cobol/pay/payroll.cbl'. The path where the file is actually found is displayed in the precompiler messages. On OS/2 and Windows platforms, substitute back slashes (\) for the forward slashes in the above example.

**Note:** The setting of DB2INCLUDE is cached by the DB2 Command Line Processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

- To continue a string constant to the next line, column 7 of the continuing line must contain a '-' and column 12 or beyond must contain a string delimiter.
- SQL arithmetic operators must be delimited by blanks.
- Full-line COBOL comments can occur anywhere in the program, including within SQL statements.
- Use host variables exactly as declared when referencing host variables within an SQL statement.
- Substitution of white space characters such as end-of-line and TAB characters occur as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.

- When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a COBOL program. TABs are not modified.

Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, OS/2 uses Carriage Return/Line Feed for end-of-line, whereas UNIX-based systems use just a Line Feed.

---

## Host Variables

Host variables are COBOL language variables that are referenced within SQL statements. They allow an application to pass input data to the database manager and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other COBOL variable. Obey the rules described below when naming, declaring, and using host variables.

### Naming Host Variables

The SQL precompiler identifies host variables by their declared name. The following rules apply:

- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than SQL, sql, DB2, or db2, which are reserved for system use.
- FILLER items using the declaration syntaxes described below are permitted in group host variable declarations, and will be ignored by the precompiler. However, if you use FILLER more than once within an SQL DECLARE section, the precompiler fails. You may not include FILLER items in VARCHAR, LONG VARCHAR, VARGRAPHIC or LONG VARGRAPHIC declarations.
- You can use hyphens in host variable names.  
SQL interprets a hyphen enclosed by spaces as a subtraction operator. Use hyphens without spaces in host variable names.
- The REDEFINES clause is permitted in host variable declarations.
- Level-88 declarations are permitted in the host variable declare section, but are ignored.

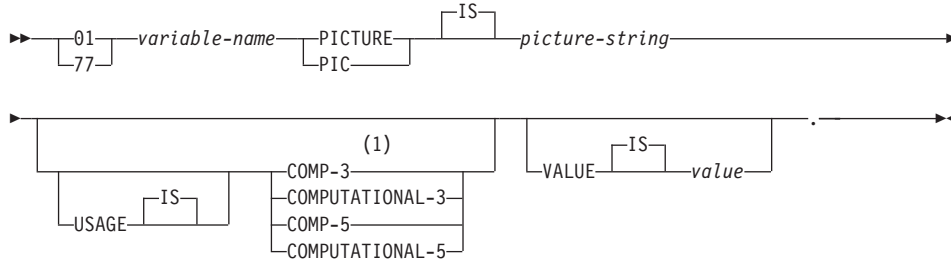
### Declaring Host Variables

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The COBOL precompiler only recognizes a subset of valid COBOL declarations.

Syntax for Numeric Host Variables in COBOL shows the syntax for numeric host variables.

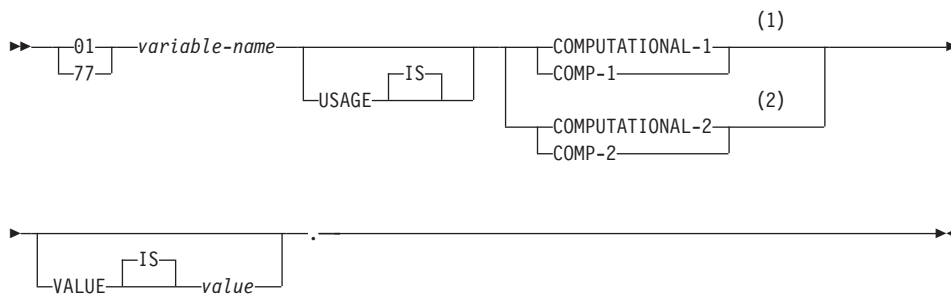
### Syntax for Numeric Host Variables in COBOL



#### Notes:

1. An alternative for COMP-3 is PACKED-DECIMAL.

### Floating Point



#### Notes:

1. REAL (SQLTYPE 480), Length 4
2. DOUBLE (SQLTYPE 480), Length 8

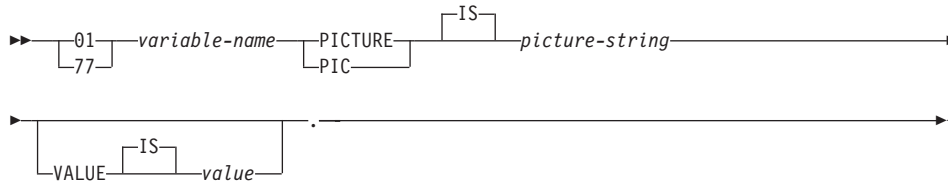
### Numeric Host Variable Considerations:

1. *Picture-string* must have one of the following forms:
  - S9(m)V9(n)
  - S9(m)V
  - S9(m)
2. Nines may be expanded (e.g., "S999" instead of S9(3))

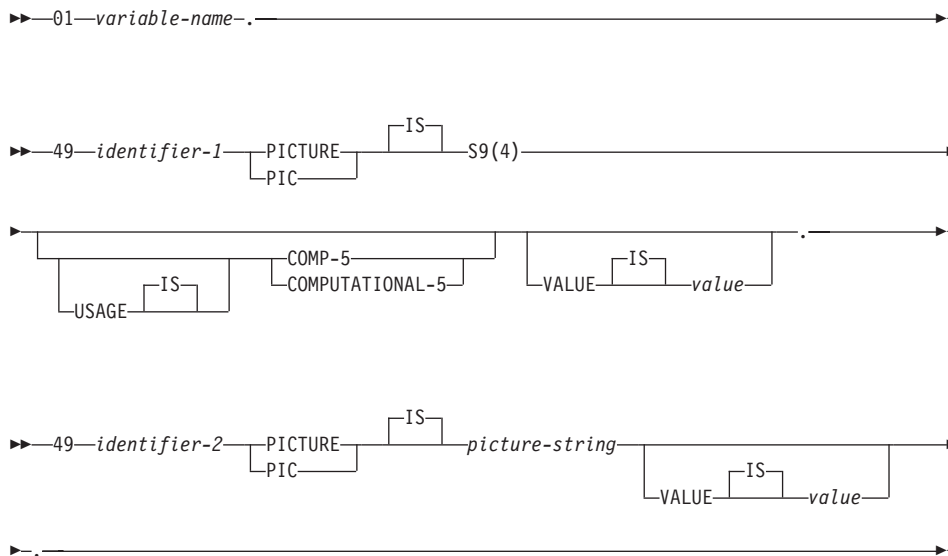
3.  $m$  and  $n$  must be positive integers.

Syntax for Character Host Variables in COBOL: Fixed Length shows the syntax for character host variables.

#### Syntax for Character Host Variables in COBOL: Fixed Length



#### Variable Length



#### Character Host Variable Consideration:

1. *Picture-string* must have the form  $X(m)$ . Alternately,  $X$ 's may be expanded (for example, "XXX" instead of "X(3)").
2.  $m$  is from 1 to 254 for fixed-length strings.
3.  $m$  is from 1 to 32 700 for variable-length strings.
4. If  $m$  is greater than 32 672, the host variable will be treated as a LONG VARCHAR string, and its use may be restricted.

5. Use X and 9 as the picture characters in any PICTURE clause. Other characters are not allowed.
6. Variable-length strings consist of a length item and a value item. You can use acceptable COBOL names for the length item and the string item. However, refer to the variable-length string by the collective name in SQL statements.
7. In a CONNECT statement, such as shown below, COBOL character string host variables dbname and userid will have any trailing blanks removed before processing:

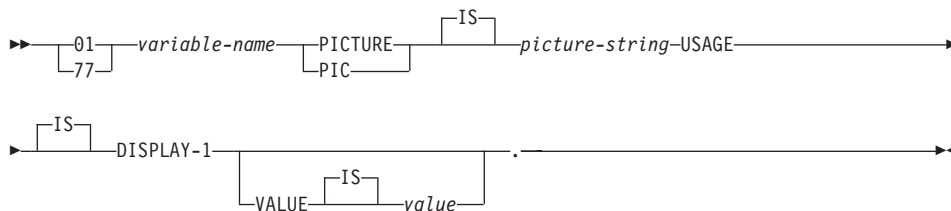
```
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

However, because blanks can be significant in passwords, the p-word host variable should be declared as a VARCHAR data item, so that your application can explicitly indicate the significant password length for the CONNECT statement as follows:

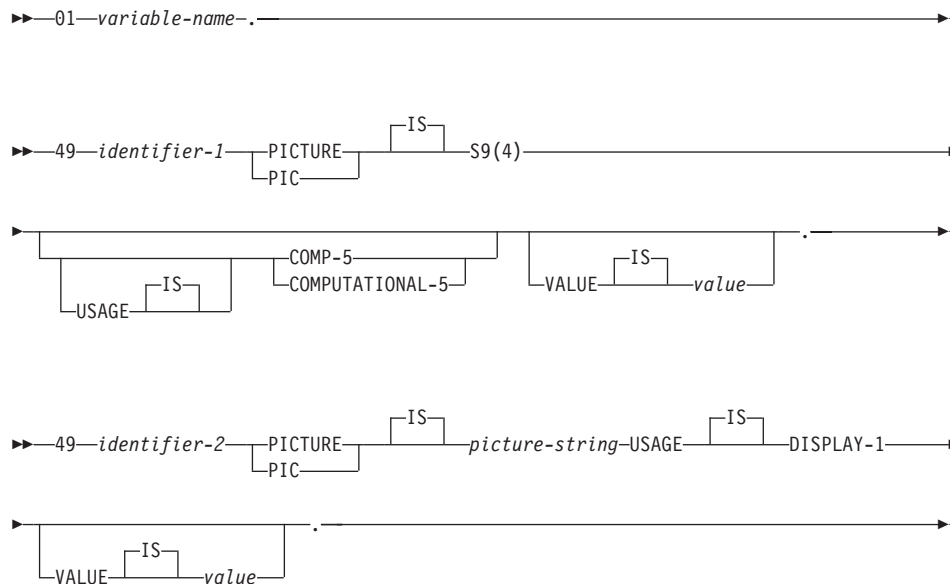
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 dbname PIC X(8).
01 userid PIC X(8).
01 p-word.
 49 L PIC S9(4) COMP-5.
 49 D PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
 MOVE "sample" TO dbname.
 MOVE "userid" TO userid.
 MOVE "password" TO D OF p-word.
 MOVE 8 TO L OF p-word.
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

Syntax for Graphic Host Variables in COBOL: Fixed Length shows the syntax for graphic host variables.

#### Syntax for Graphic Host Variables in COBOL: Fixed Length



#### Variable Length



### Graphic Host Variable Considerations:

1. *Picture-string* must have the form *G(m)*. Alternately, G's may be expanded (for example, "GGG" instead of "G(3)").
2. *m* is from 1 to 127 for fixed-length strings.
3. *m* is from 1 to 16 350 for variable-length strings.
4. If *m* is greater than 16 336, the host variable will be treated as a LONG VARGRAPHIC string, and its use may be restricted.

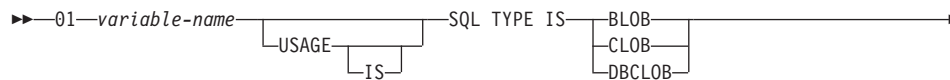
### Indicator Variables

Indicator variables should be declared as a PIC S9(4) COMP-5 data type.

### LOB Declarations in COBOL

Syntax for LOB Host Variables in COBOL shows the syntax for declaring large object (LOB) host variables in COBOL.

#### Syntax for LOB Host Variables in COBOL





### LOB Host Variable Considerations:

1. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
2. For DBCLOB 1 <= lob-length <= 1 073 741 823.
3. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G can be in either uppercase, lowercase, or mixed.
4. Initialization within the LOB declaration is not permitted.
5. The host variable name prefixes LENGTH and DATA in the precompiler generated code.

### BLOB Example:

Declaring:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

Results in the generation of the following structure:

```
01 MY-BLOB.
49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
49 MY-BLOB-DATA PIC X(2097152).
```

### CLOB Example:

Declaring:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

Results in the generation of the following structure:

```
01 MY-CLOB.
49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
49 MY-CLOB-DATA PIC X(131072000).
```

### DBCLOB Example:

Declaring:

```
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

Results in the generation of the following structure:

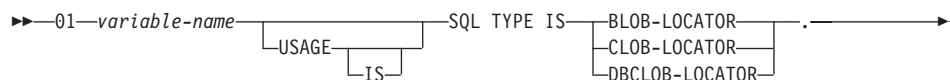
```
01 MY-DBCLOB.
49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```



## LOB Locator Declarations in COBOL

Syntax for LOB Locator Host Variables in COBOL shows the syntax for declaring large object (LOB) locator host variables in COBOL.

### Syntax for LOB Locator Host Variables in COBOL



### LOB Locator Host Variable Considerations:

1. SQL TYPE IS, BLOB-LOCATOR, CLOB-LOCATOR, DBCLOB-LOCATOR can be either uppercase, lowercase, or mixed.
2. Initialization of locators is not permitted.

**BLOB Locator Example** (other LOB locator types are similar):

Declaring:

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

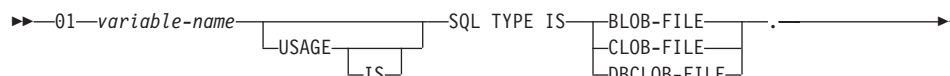
Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

## File Reference Declarations in COBOL

Syntax for File Reference Host Variables in COBOL shows the syntax for declaring file reference host variables in COBOL.

### Syntax for File Reference Host Variables in COBOL



- SQL TYPE IS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE can be either uppercase, lowercase, or mixed.

**BLOB File Reference Example** (other LOB types are similar):

Declaring:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

Results in the generation of the following declaration:

```

01 MY-FILE.
 49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
 49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
 49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
 49 MY-FILE-NAME PIC X(255).

```

## Host Structure Support

The COBOL precompiler supports declarations of group data items in the host variable declare section. Among other things, this provides a shorthand for referring to a set of elementary data items in an SQL statement. For example, the following group data item can be used to access some of the columns in the STAFF table of the SAMPLE database:

```

01 staff-record.
 05 staff-id pic s9(4) comp-5.
 05 staff-name.
 49 l pic s9(4) comp-5.
 49 d pic x(9).
 05 staff-info.
 10 staff-dept pic s9(4) comp-5.
 10 staff-job pic x(5).

```

Group data items in the declare section can have any of the valid host variable types described above as subordinate data items. This includes all numeric and character types, as well as all large object types. You can nest group data items up to 10 levels. Note that you must declare VARCHAR character types with the subordinate items at level 49, as in the above example. If they are not at level 49, the VARCHAR is treated as a group data item with two subordinates, and is subject to the rules of declaring and using group data items. In the example above, staff-info is a group data item, whereas staff-name is a VARCHAR. The same principle applies to LONG VARCHAR, VARGRAPHIC and LONG VARGRAPHIC. You may declare group data items at any level between 02 and 49.

You can use group data items and their subordinates in four ways:

### Method 1.

The entire group may be referenced as a single host variable in an SQL statement:

```

EXEC SQL SELECT id, name, dept, job
 INTO :staff-record
 FROM staff WHERE id = 10 END-EXEC.

```

The precompiler converts the reference to staff-record into a list, separated by commas, of all the subordinate items declared within staff-record. Each elementary item is qualified with the group names of all levels to prevent naming conflicts with other items. This is equivalent to the following method.

## Method 2.

The second way of using group data items:

```
EXEC SQL SELECT id, name, dept, job
 INTO
 :staff-record.staff-id,
 :staff-record.staff-name,
 :staff-record.staff-info.staff-dept,
 :staff-record.staff-info.staff-job
 FROM staff WHERE id = 10 END-EXEC.
```

**Note:** The reference to staff-id is qualified with its group name using the prefix staff-record., and not staff-id of staff-record as in pure COBOL.

Assuming there are no other host variables with the same names as the subordinates of staff-record, the above statement can also be coded as in method 3, eliminating the explicit group qualification.

## Method 3.

Here, subordinate items are referenced in a typical COBOL fashion, without being qualified to their particular group item:

```
EXEC SQL SELECT id, name, dept, job
 INTO
 :staff-id,
 :staff-name,
 :staff-dept,
 :staff-job
 FROM staff WHERE id = 10 END-EXEC.
```

As in pure COBOL, this method is acceptable to the precompiler as long as a given subordinate item can be uniquely identified. If, for example, staff-job occurs in more than one group, the precompiler issues an error indicating an ambiguous reference:

```
SQL0088N Host variable "staff-job" is ambiguous.
```

## Method 4.

To resolve the ambiguous reference, you can use partial qualification of the subordinate item, for example:

```
EXEC SQL SELECT id, name, dept, job
 INTO
 :staff-id,
 :staff-name,
 :staff-info.staff-dept,
 :staff-info.staff-job
 FROM staff WHERE id = 10 END-EXEC.
```

Because a reference to a group item alone, as in method 1, is equivalent to a comma-separated list of its subordinates, there are instances where this type of reference leads to an error. For example:

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

Here, the CONNECT statement expects a single character-based host variable. By giving the staff-record group data item instead, the host variable results in the following precompile-time error:

```
SQL0087N Host variable "staff-record" is a structure used where
structure references are not permitted.
```

Other uses of group items which cause an SQL0087N to occur include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables, and SQLDA references. Groups with only one subordinate are permitted in such situations, as are references to individual subordinates, as in method 2, 3 and 4 above.

## Indicator Tables

The COBOL precompiler supports the declaration of tables of indicator variables, which are convenient to use with group data items. They are declared as follows:

```
01 <indicator-table-name>.
 05 <indicator-name> pic s9(4) comp-5
 occurs <table-size> times.
```

For example:

```
01 staff-indicator-table.
 05 staff-indicator pic s9(4) comp-5
 occurs 7 times.
```

This indicator table can be used effectively with the first format of group item reference above:

```
EXEC SQL SELECT id, name, dept, job
INTO :staff-record :staff-indicator
FROM staff WHERE id = 10 END-EXEC.
```

Here, the precompiler detects that staff-indicator was declared as an indicator table, and expands it into individual indicator references when it processes the SQL statement. staff-indicator(1) is associated with staff-id of staff-record, staff-indicator(2) is associated with staff-name of staff-record, and so on.

**Note:** If there are  $k$  more indicator entries in the indicator table than there are subordinates in the data item (for example, if staff-indicator has 10 entries, making  $k=6$ ), the  $k$  extra entries at the end of the indicator table are ignored. Likewise, if there are  $k$  fewer indicator entries than subordinates, the last  $k$  subordinates in the group item do not have

indicators associated with them. **Note that you can refer to individual elements in an indicator table in an SQL statement.**

## Using REDEFINES in COBOL Group Data Items

You can use the REDEFINES clause when declaring host variables. If you declare a member of a group data item with the REDEFINES clause and that group data item is referred to as a whole in an SQL statement, any subordinate items containing the REDEFINES clause are not expanded. For example:

```
01 foo.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

Referring to foo in an SQL statement as follows:

```
... INTO :foo ...
```

The above statement is equivalent to:

```
... INTO :foo.a, :foo.b ...
```

That is, the subordinate item a1, declared with the REDEFINES clause is not automatically expanded out in such situations. If a1 is unambiguous, you can explicitly refer to a subordinate with a REDEFINES clause in an SQL statement, as follows:

```
... INTO :foo.a1 ...
```

or

```
... INTO :a1 ...
```

## Using BINARY/COMP-4 COBOL Data Types

The DB2 COBOL precompiler supports the use of BINARY, COMP, and COMP-4 data types wherever integer host variables and indicators are permitted, as long as the target COBOL compiler views (or can be made to view) the BINARY, COMP, or COMP-4 data types as equivalent to the COMP-5 data type. In this book, such host variables and indicators are shown with type COMP-5. Target compilers supported by DB2 that treat COMP, COMP-4, BINARY COMP and COMP-5 as equivalent are:

- IBM COBOL Set for AIX
- Micro Focus COBOL for AIX
- IBM COBOL Visual Set for OS/2 (with the -qbinary(native) option set)
- IBM VisualAge for COBOL for OS/2, Windows NT and Windows 95, (with the -qbinary(native) option set)

---

## Supported SQL Data Types

Certain predefined COBOL data types correspond to column types. Only these COBOL data types can be declared as host variables.

Table 22 shows the COBOL equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Not every possible data description for host variables is recognized. COBOL data items must be consistent with the ones described in the following table. If you use other data items, an error can result.

**Note:** There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 22. SQL Data Types Mapped to COBOL Declarations

SQL Column Type <sup>1</sup>	COBOL Data Type	SQL Column Type Description
SMALLINT (500 or 501)	01 name PIC S9(4) COMP-5.	16-bit signed integer
INTEGER (496 or 497)	01 name PIC S9(9) COMP-5.	32-bit signed integer
BIGINT (492 or 493)	01 name PIC S9(18) COMP-5.	64-bit signed integer
DECIMAL( <i>p,s</i> ) (484 or 485)	01 name PIC S9( <i>m</i> )V9( <i>n</i> ) COMP-3.	Packed decimal
REAL <sup>2</sup> (480 or 481)	01 name USAGE IS COMP-1.	Single-precision floating point
DOUBLE <sup>3</sup> (480 or 481)	01 name USAGE IS COMP-2.	Double-precision floating point
CHAR( <i>n</i> ) (452 or 453)	01 name PIC X( <i>n</i> ).	Fixed-length character string
VARCHAR( <i>n</i> ) (448 or 449)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC X( <i>n</i> ).  1<= <i>n</i> <=32 672	Variable-length character string
LONG VARCHAR (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X( <i>n</i> ).  32 673<= <i>n</i> <=32 700	Long variable-length character string

Table 22. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type <sup>1</sup>	COBOL Data Type	SQL Column Type Description
CLOB(n) (408 or 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n).  1<=n<=2 147 483 647	Large object variable-length character string
CLOB locator variable <sup>4</sup> (964 or 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	Identifies CLOB entities residing on the server
CLOB file reference variable <sup>4</sup> (808 or 809)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
BLOB(n) (404 or 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n).  1<=n<=2 147 483 647	Large object variable-length binary string
BLOB locator variable <sup>4</sup> (960 or 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	Identifies BLOB entities residing on the server
BLOB file reference variable <sup>4</sup> (804 or 805)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	Descriptor for file containing CLOB data
DATE (384 or 385)	01 identifier PIC X(10).	10-byte character string
TIME (388 or 389)	01 identifier PIC X(8).	8-byte character string
TIMESTAMP (392 or 393)	01 identifier PIC X(26).	26-byte character string
<b>Note:</b> The following data types are only available in the DBCS environment.		
GRAPHIC(n) (468 or 469)	01 name PIC G(n) DISPLAY-1.	Fixed-length double-byte character string
VARGRAPHIC(n) (464 or 465)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(n) DISPLAY-1.  1<=n<=16 336	Variable length double-byte character string with 2-byte string length indicator
LONG VARGRAPHIC (472 or 473)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(n) DISPLAY-1.  16 337<=n<=16 350	Variable length double-byte character string with 2-byte string length indicator
DBCLOB(n) (412 or 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n).  1<=n<=1 073 741 823	Large object variable length double-byte character string with 4-byte string length indicator
DBCLOB locator variable <sup>4</sup> (968 or 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable <sup>4</sup> (812 or 813)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	Descriptor for file containing DBCLOB data

Table 22. SQL Data Types Mapped to COBOL Declarations (continued)

SQL Column Type <sup>1</sup>	COBOL Data Type	SQL Column Type Description
<b>Note:</b>		
1. The first number under <b>SQL Column Type</b> indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.		
2. FLOAT( <i>n</i> ) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).		
3. The following SQL types are synonyms for DOUBLE:		
<ul style="list-style-type: none"> <li>FLOAT</li> <li>FLOAT(<i>n</i>) where <math>24 &lt; n &lt; 54</math> is</li> <li>DOUBLE PRECISION</li> </ul>		
4. This is not a column type but a host variable type.		

The following is a sample SQL declare section with a host variable declared for each supported SQL data type.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
01 age PIC S9(4) COMP-5.
01 divis PIC S9(9) COMP-5.
01 salary PIC S9(6)V9(3) COMP-3.
01 bonus USAGE IS COMP-1.
01 wage USAGE IS COMP-2.
01 nm PIC X(5).
01 varchar.
 49 leng PIC S9(4) COMP-5.
 49 strg PIC X(14).
01 longvchar.
 49 len PIC S9(4) COMP-5.
 49 str PIC X(6027).
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).
01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).
01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.
01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).
01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.
01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.
01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.
01 dt PIC X(10).
01 tm PIC X(8).
01 tmstp PIC X(26).
01 wage-ind PIC S9(4) COMP-5.
*
EXEC SQL END DECLARE SECTION END-EXEC.
```

The following are additional rules for supported COBOL data types:



- PIC S9 and COMP-3/COMP-5 are required where shown.
- You can use level number 77 instead of 01 for all column types except VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC and all LOB variable types.
- Use the following rules when declaring host variables for DECIMAL(p,s) column types. Refer to the following sample:
 

```
01 identifier PIC S9(m)V9(n) COMP-3
```

  - Use V to denote the decimal point.
  - Values for *n* and *m* must be greater than or equal to 1.
  - The value for *n* + *m* cannot exceed 31.
  - The value for *s* equals the value for *n*.
  - The value for *p* equals the value for *n* + *m*.
  - The repetition factors (*n*) and (*m*) are optional. The following examples are all valid:
 

```
01 identifier PIC S9(3)V COMP-3
01 identifier PIC SV9(3) COMP-3
01 identifier PIC S9V COMP-3
01 identifier PIC SV9 COMP-3
```
  - PACKED-DECIMAL can be used instead of COMP-3.
- Arrays are **not** supported by the COBOL precompiler:

## FOR BIT DATA

Certain database columns can be declared FOR BIT DATA. These columns, which generally contain characters, are used to hold binary information. The CHAR(*n*), VARCHAR, LONG VARCHAR, and BLOB data types are the COBOL host variable types that can contain binary data. Use these data types when working with columns with the FOR BIT DATA attribute.

---

## SQLSTATE and SQLCODE Variables

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PICTURE X(5).
01 SQLCODE PICTURE S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. The '01' can also be '77' and the 'PICTURE' can be 'PIC'. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications made up of multiple source files, the SQLCODE and SQLSTATE declarations may be included in each source file as shown above.

---

## Japanese or Traditional Chinese EUC, and UCS-2 Considerations

Any graphic data sent from your application running under an eucJP or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is responsible for converting to and from UCS-2 since this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. DB2 Universal Database does not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

For further information on these functions, refer to the *SQL Reference*. For general EUC application development guidelines, see “Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations” on page 419.

---

## Object Oriented COBOL

If you are using Object Oriented COBOL, you must observe the following:

- SQL statements can only appear in the first program or class in a compile unit. This is because the precompiler inserts temporary working data into the first Working-Storage section it sees.
- In an Object Oriented COBOL program, every class containing SQL statements must have a class-level Working-Storage Section, even if it is empty. This section is used to store data definitions generated by the precompiler.

---

## Chapter 19. Programming in FORTRAN

Programming Considerations . . . . .	591	Declaring Host Variables . . . . .	597
Language Restrictions . . . . .	591	Indicator Variables . . . . .	600
Call by Reference . . . . .	591	LOB Declarations in FORTRAN . . . . .	600
Debugging and Comment Lines. . . . .	592	LOB Locator Declarations in FORTRAN . . . . .	601
Precompiling Considerations . . . . .	592	File Reference Declarations in FORTRAN . . . . .	601
Input and Output Files. . . . .	592	Supported SQL Data Types . . . . .	602
Include Files . . . . .	592	SQLSTATE and SQLCODE Variables . . . . .	604
Including Files . . . . .	595	Considerations for Multi-byte Character Sets . . . . .	605
Embedding SQL Statements . . . . .	595	Japanese or Traditional Chinese EUC, and . . . . .	
Host Variables. . . . .	597	UCS-2 Considerations . . . . .	605
Naming Host Variables. . . . .	597		

---

### Programming Considerations

Special host-language programming considerations are discussed in the following pages. Included is information on language restrictions, host language specific include files, embedding SQL statements, host variables, and supported data types for host variables.

**Note:** FORTRAN support stabilized in DB2 Version 5, and no enhancements for FORTRAN support are planned for the future. For example, the FORTRAN precompiler cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 after Version 5, such as table names from 19 to 128 bytes long, you must not write your applications in FORTRAN.

---

### Language Restrictions

The following sections describe the FORTRAN language restrictions.

#### Call by Reference

Some API parameters require addresses rather than values in the call variables. The database manager provides the GET ADDRESS, DEREFERENCE ADDRESS, and COPY MEMORY APIs which simplify your ability to provide these parameters. Refer to the *Administrative API Reference* for a full description of these APIs.

## Debugging and Comment Lines

Some FORTRAN compilers treat lines with a 'D' or 'd' in column 1 as conditional lines. These lines can either be compiled for debugging or treated as comments. The precompiler will always treat lines with a 'D' or 'd' in column 1 as comments.

## Precompiling Considerations

The following items affect the precompiling process:

- The precompiler allows only digits, blanks, and tab characters within columns 1-5 on continuation lines.
- Hollerith constants are not supported in .sqf source files.

Refer to the *Application Building Guide* for information on any other precompiling considerations that may affect you.

---

## Input and Output Files

By default, the input file has an extension of .sqf, but if you use the TARGET precompile option the input file can have any extension you prefer.

By default, the output file has an extension of .f on UNIX-platforms, and .for on OS/2 and Windows-based platforms, however you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

---

## Include Files

The host-language specific include files for FORTRAN have the file extension .f on UNIX platforms, and .for on OS/2. You can use the following FORTRAN include files in your applications.

**SQL (sql.f)**      This file includes language-specific prototypes for the binder, precompiler, and error message retrieval APIs. It also defines system constants.

**SQLAPREP (sqlaprep.f)**  
This file contains definitions required to write your own precompiler.

**SQLCA (sqlca\_cn.f, sqlca\_cs.f)**  
This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the

database manager to provide an application with error information about the execution of SQL statements and API calls.

Two SQLCA files are provided for FORTRAN applications. The default, `sqlca_cs.f`, defines the SQLCA structure in an IBM SQL compatible format. The `sqlca_cn.f` file, precompiled with the SQLCA NONE option, defines the SQLCA structure for better performance.

**SQLCA\_92 (`sqlca_92.f`)**

This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of either the `sqlca_cn.f` or the `sqlca_cs.f` files when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.f` file is automatically included by the DB2 precompiler when the `LANGLEVEL` precompiler option is set to `SQL92E`.

**SQLCODES (`sqlcodes.f`)**

This file defines constants for the `SQLCODE` field of the SQLCA structure.

**SQLDA (`sqldact.f`)**

This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager. See “Allocating an SQLDA Structure” on page 152 for details of how to code an SQLDA in a FORTRAN program.

**SQLCAU (`sqleau.f`)**

This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

**SQLENV (`sqlenv.f`)**

This file defines language-specific calls for the database environment APIs, and the structures, constants, and return codes for those interfaces.

**SQL819A (`sqle819a.f`)**

If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the `CREATE DATABASE` API.

**SQL819B (sqle819b.f)**

If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQL850A (sqle850a.f)**

If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL850B (sqle850b.f)**

If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQL932A (sqle932a.f)**

If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL932B (sqle932b.f)**

If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252A (sql1252a.f)**

If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252B (sql1252b.f)**

If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLMON (sqlmon.f)**

This file defines language-specific calls for the database system monitor APIs, and the structures, constants, and return codes for those interfaces.

**SQLSTATE (sqlstate.f)**

This file defines constants for the SQLSTATE field of the SQLCA structure.

**SQLUTIL (sqlutil.f)**

This file defines the language-specific calls for the utility APIs, and the structures, constants, and codes required for those interfaces.

**Including Files**

There are two methods for including files: the EXEC SQL INCLUDE statement and the FORTRAN INCLUDE statement. The precompiler will ignore FORTRAN INCLUDE statements, and only process files included with the EXEC SQL statement.

To locate the INCLUDE file, the DB2 FORTRAN precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- EXEC SQL INCLUDE payroll

If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the precompiler searches for payroll.sqf, then payroll.f (payroll.for on OS/2) in each directory in which it looks.

- EXEC SQL INCLUDE 'pay/payroll.f'

If the file name is enclosed in quotation marks, as above, no extension is added to the name. (For OS/2, the file would be specified as 'pay\payroll.for'.)

If the file name in quotation marks does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for AIX, if DB2INCLUDE is set to '/disk2:myfiles/fortran', the precompiler searches for './pay/payroll.f', then '/disk2/pay/payroll.f', and finally './myfiles/cobol/pay/payroll.f'. The path where the file is actually found is displayed in the precompiler messages. On OS/2, substitute back slashes (\) for the forward slashes, and substitute 'for' for the 'f' extension in the above example.

**Note:** The setting of DB2INCLUDE is cached by the DB2 Command Line Processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

---

**Embedding SQL Statements**

Embedded SQL statements consist of the following three elements:

<b>Element</b>	<b>Correct FORTRAN Syntax</b>
<b>Keyword</b>	EXEC SQL
<b>Statement string</b>	Any valid SQL statement with blanks as delimiters
<b>Statement terminator</b>	End of source line.

The end of the source line serves as the statement terminator. If the line is continued, the statement terminator is the end of the last continued line.

For example:

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

The following rules apply to embedded SQL statements:

- Code SQL statements between columns 7 and 72 only.
- Use full-line FORTRAN comments, or SQL comments, but do not use the FORTRAN end-of-line comment '!' character in SQL statements. This comment character may be used elsewhere, including host variable declarations.
- Use blanks as delimiters when coding embedded SQL statements, even though FORTRAN statements do not require blanks as delimiters.
- Use only one SQL statement for each FORTRAN source line. Normal FORTRAN continuation rules apply for statements that require more than one source line. Do not split the EXEC SQL keyword pair between lines.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (--) followed by a string of zero or more characters and terminated by a line end.
- FORTRAN comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
  - The extension of using ! to code a FORTRAN comment at the end of a line is not supported within an embedded SQL statement.
- Use exponential notation when specifying a real constant in SQL statements. The database manager interprets a string of digits with a decimal point in an SQL statement as a decimal constant, not a real constant.
- Statement numbers are invalid on SQL statements that precede the first executable FORTRAN statement. If an SQL statement has a statement number associated with it, the precompiler generates a labeled CONTINUE statement that directly precedes the SQL statement.



- Use host variables exactly as declared when referencing host variables within an SQL statement.
- Substitution of white space characters such as end-of-line and TAB characters occur as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a FORTRAN program. TABs are not modified.

Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, OS/2 uses Carriage Return/Line Feed for end-of-line, whereas UNIX-based systems use just a Line Feed.

---

## Host Variables

Host variables are FORTRAN language variables that are referenced within SQL statements. They allow an application to pass input data to the database manager and receive output data from it. After the application is precompiled, host variables are used by the compiler as any other FORTRAN variable. Use the following suggestions when naming, declaring, and using host variables.

### Naming Host Variables

The SQL precompiler identifies host variables by their declared name. The following suggestions apply:

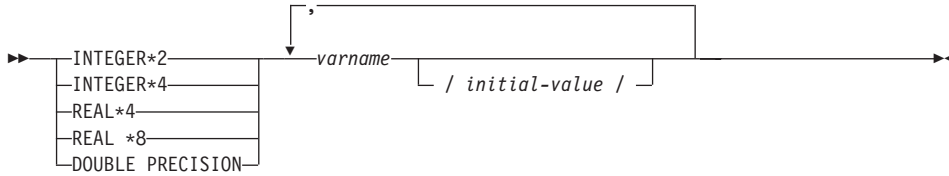
- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than SQL, sql, DB2, or db2, which are reserved for system use.

### Declaring Host Variables

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The FORTRAN precompiler only recognizes a subset of valid FORTRAN declarations as valid host variable declarations. These declarations define either numeric or character variables. A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time or timestamp SQL input or output value. The programmer must ensure that output variables are long enough to contain the values that they will receive. Syntax for Numeric Host Variables in FORTRAN shows the syntax for numeric host variables.

## Syntax for Numeric Host Variables in FORTRAN

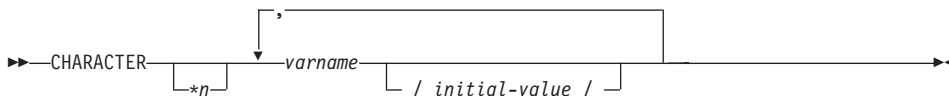


### Numeric Host Variable Considerations:

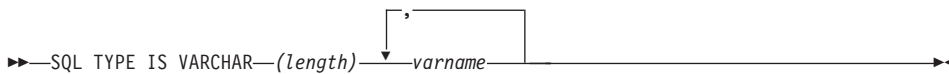
1. REAL\*8 and DOUBLE PRECISION are equivalent.
2. Use an E rather than a D as the exponent indicator for REAL\*8 constants.

Syntax for Character Host Variables in FORTRAN: Fixed Length shows the syntax for character host variables.

### Syntax for Character Host Variables in FORTRAN: Fixed Length



### Variable Length



### Character Host Variable Considerations:

1. \*n has a maximum value of 254.
2. When length is between 1 and 32 672 inclusive, the host variable has type VARCHAR(SQLTYPE 448).
3. When length is between 32 673 and 32 700 inclusive, the host variable has type LONG VARCHAR(SQLTYPE 456).
4. Initialization of VARCHAR and LONG VARCHAR host variables is not permitted within the declaration.

### VARCHAR Example:

Declaring:

```
sql type is varchar(1000) my_varchar
```

Results in the generation of the following structure:

```

character my_varchar(1000+2)
integer*2 my_varchar_length
character my_varchar_data(1000)
equivalence(my_varchar(1),
+ my_varchar_length)
equivalence(my_varchar(3),
+ my_varchar_data)

```

The application may manipulate both `my_varchar_length` and `my_varchar_data`; for example, to set or examine the contents of the host variable. The base name (in this case, `my_varchar`), is used in SQL statements to refer to the VARCHAR as a whole.

### LONG VARCHAR Example:

Declaring:

```
sql type is varchar(10000) my_lvarchar
```

Results in the generation of the following structure:

```

character my_lvarchar(10000+2)
integer*2 my_lvarchar_length
character my_lvarchar_data(10000)
equivalence(my_lvarchar(1),
+ my_lvarchar_length)
equivalence(my_lvarchar(3),
+ my_lvarchar_data)

```

The application may manipulate both `my_lvarchar_length` and `my_lvarchar_data`; for example, to set or examine the contents of the host variable. The base name (in this case, `my_lvarchar`), is used in SQL statements to refer to the LONG VARCHAR as a whole.

**Note:** In a CONNECT statement, such as in the following example, FORTRAN character string host variables `dbname` and `userid` will have any trailing blanks removed before processing.

```
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

However, because blanks can be significant in passwords, you should declare host variables for passwords as VARCHAR, and have the length field set to reflect the actual password length:

```

EXEC SQL BEGIN DECLARE SECTION
 character*8 dbname, userid
 sql type is varchar(18) passwd
EXEC SQL END DECLARE SECTION
character*18 passwd_string
equivalence(passwd_data,passwd_string)
dbname = 'sample'
userid = 'userid'

```

```

passwd_length= 8
passwd_string = 'password'
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd

```

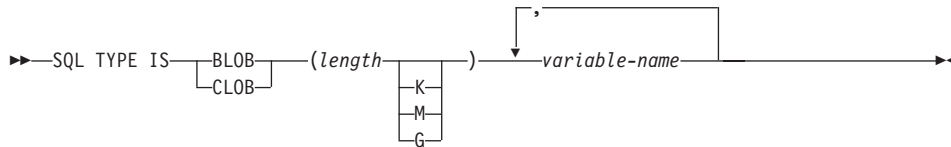
## Indicator Variables

Indicator variables should be declared as an INTEGER\*2 data type.

## LOB Declarations in FORTRAN

Syntax for Large Object (LOB) Host Variables in FORTRAN shows the syntax for declaring large object (LOB) host variables in FORTRAN.

### Syntax for Large Object (LOB) Host Variables in FORTRAN



### LOB Host Variable Considerations:

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB, CLOB, K, M, G can be in either uppercase, lowercase, or mixed.
3. For BLOB and CLOB  $1 \leq \text{lob-length} \leq 2\,147\,483\,647$ .
4. The initialization of a LOB within a LOB declaration is not permitted.
5. The host variable name prefixes 'length' and 'data' in the precompiler generated code.

### BLOB Example:

Declaring:

```
sql type is blob(2m) my_blob
```

Results in the generation of the following structure:

```

character my_blob(2097152+4)
integer*4 my_blob_length
character my_blob_data(2097152)
equivalence(my_blob(1),
+ my_blob_length)
equivalence(my_blob(5),
+ my_blob_data)

```

### CLOB Example:

Declaring:

```
sql type is clob(125m) my_clob
```

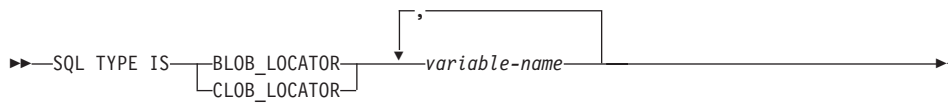
Results in the generation of the following structure:

```
character my_clob(131072000+4)
integer*4 my_clob_length
character my_clob_data(131072000)
equivalence(my_clob(1),
+ my_clob_length)
equivalence(my_clob(5),
+ my_clob_data)
```

## LOB Locator Declarations in FORTRAN

Syntax for Large Object (LOB) Locator Host Variables in FORTRAN shows the syntax for declaring large object (LOB) locator host variables in FORTRAN.

### Syntax for Large Object (LOB) Locator Host Variables in FORTRAN



### LOB Locator Host Variable Considerations:

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB\_LOCATOR, CLOB\_LOCATOR can be either uppercase, lowercase, or mixed.
3. Initialization of locators is not permitted.

### CLOB Locator Example (BLOB locator is similar):

Declaring:

```
SQL TYPE IS CLOB_LOCATOR my_locator
```

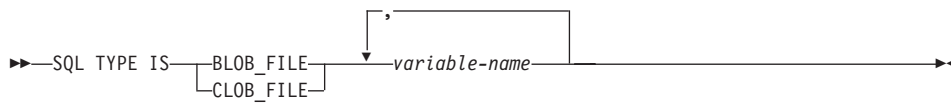
Results in the generation of the following declaration:

```
integer*4 my_locator
```

## File Reference Declarations in FORTRAN

Syntax for File Reference Host Variables in FORTRAN shows the syntax for declaring file reference host variables in FORTRAN.

### Syntax for File Reference Host Variables in FORTRAN



#### File Reference Host Variable Considerations:

1. Graphic types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB\_FILE, CLOB\_FILE can be either uppercase, lowercase, or mixed.

**Example of a BLOB file reference variable** (CLOB file reference variable is similar):

```
SQL TYPE IS BLOB_FILE my_file
```

Results in the generation of the following declaration:

```
character my_file(267)
integer*4 my_file_name_length
integer*4 my_file_data_length
integer*4 my_file_file_options
character*255 my_file_name
equivalence(my_file(1),
+ my_file_name_length)
equivalence(my_file(5),
+ my_file_data_length)
equivalence(my_file(9),
+ my_file_file_options)
equivalence(my_file(13),
+ my_file_name)
```

---

## Supported SQL Data Types

Certain predefined FORTRAN data types correspond to database manager column types. Only these FORTRAN data types can be declared as host variables.

Table 23 on page 603 shows the FORTRAN equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

**Note:** There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 23. SQL Data Types Mapped to FORTRAN Declarations

SQL Column Type <sup>1</sup>	FORTTRAN Data Type	SQL Column Type Description
SMALLINT (500 or 501)	INTEGER*2	16-bit, signed integer
INTEGER (496 or 497)	INTEGER*4	32-bit, signed integer
REAL <sup>2</sup> (480 or 481)	REAL*4	Single precision floating point
DOUBLE <sup>3</sup> (480 or 481)	REAL*8	Double precision floating point
DECIMAL( <i>p,s</i> ) (484 or 485)	No exact equivalent; use REAL*8	Packed decimal
CHAR( <i>n</i> ) (452 or 453)	CHARACTER* <i>n</i>	Fixed-length character string of length <i>n</i> where <i>n</i> is from 1 to 254
VARCHAR( <i>n</i> ) (448 or 449)	SQL TYPE IS VARCHAR( <i>n</i> ) where <i>n</i> is from 1 to 32 672	Variable-length character string
LONG VARCHAR (456 or 457)	SQL TYPE IS VARCHAR( <i>n</i> ) where <i>n</i> is from 32 673 to 32 700	Long variable-length character string
CLOB( <i>n</i> ) (408 or 409)	SQL TYPE IS CLOB ( <i>n</i> ) where <i>n</i> is from 1 to 2 147 483 647	Large object variable-length character string
CLOB locator variable <sup>4</sup> (964 or 965)	SQL TYPE IS CLOB_LOCATOR	Identifies CLOB entities residing on the server
CLOB file reference variable <sup>4</sup> (808 or 809)	SQL TYPE IS CLOB_FILE	Descriptor for file containing CLOB data
BLOB( <i>n</i> ) (404 or 405)	SQL TYPE IS BLOB( <i>n</i> ) where <i>n</i> is from 1 to 2 147 483 647	Large object variable-length binary string
BLOB locator variable <sup>4</sup> (960 or 961)	SQL TYPE IS BLOB_LOCATOR	Identifies BLOB entities on the server
BLOB file reference variable <sup>4</sup> (804 or 805)	SQL TYPE IS BLOB_FILE	Descriptor for the file containing BLOB data
DATE (384 or 385)	CHARACTER*10	10-byte character string
TIME (388 or 389)	CHARACTER*8	8-byte character string
TIMESTAMP (392 or 393)	CHARACTER*26	26-byte character string

Table 23. SQL Data Types Mapped to FORTRAN Declarations (continued)

SQL Column Type <sup>1</sup>	FORTRAN Data Type	SQL Column Type Description
<b>Note:</b>		
1. The first number under <b>SQL Column Type</b> indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.		
2. FLOAT( <i>n</i> ) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).		
3. The following SQL types are synonyms for DOUBLE:		
• FLOAT		
• FLOAT( <i>n</i> ) where $24 < n < 54$ is		
• DOUBLE PRECISION		
4. This is not a column type but a host variable type.		

The following is a sample SQL declare section with a host variable declared for each supported data type:

```
EXEC SQL BEGIN DECLARE SECTION
 INTEGER*2 AGE /26/
 INTEGER*4 DEPT
 REAL*4 BONUS
 REAL*8 SALARY
 CHARACTER MI
 CHARACTER*112 ADDRESS
 SQL TYPE IS VARCHAR (512) DESCRIPTION
 SQL TYPE IS VARCHAR (32000) COMMENTS
 SQL TYPE IS CLOB (1M) CHAPTER
 SQL TYPE IS CLOB_LOCATOR CHAPLOC
 SQL TYPE IS CLOB_FILE CHAPFL
 SQL TYPE IS BLOB (1M) VIDEO
 SQL TYPE IS BLOB_LOCATOR VIDLOC
 SQL TYPE IS BLOB_FILE VIDFL
 CHARACTER*10 DATE
 CHARACTER*8 TIME
 CHARACTER*26 TIMESTAMP
 INTEGER*2 WAGE_IND
EXEC SQL END DECLARE SECTION
```

The following are additional rules for supported FORTRAN data types:

- You may define dynamic SQL statements longer than 254 characters by using VARCHAR, LONG VARCHAR, OR CLOB host variables.

### SQLSTATE and SQLCODE Variables

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:



```

EXEC SQL BEGIN DECLARE SECTION;
 CHARACTER*5 SQLSTATE
 INTEGER SQLCOD
 .
 .
 .
EXEC SQL END DECLARE SECTION

```

If neither of these is specified, the SQLCOD declaration is assumed during the precompile step. The variable named 'SQLSTATE' may also be 'SQLSTA'. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications that contain multiple source files, the declarations of SQLCOD and SQLSTATE may be included in each source file as shown above.

---

## Considerations for Multi-byte Character Sets

There are no graphic (multi-byte) host variable data types supported in FORTRAN. Only mixed character host variables are supported through the character data type. It is possible to create a user SQLDA that contains graphic data.

---

## Japanese or Traditional Chinese EUC, and UCS-2 Considerations

Any graphic data sent from your application running under an eucJP or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is responsible for converting to and from UCS-2 since this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. DB2 Universal Database does not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

For further information on these functions, refer to the *SQL Reference*.

For general EUC application development guidelines, see “Japanese and Traditional Chinese EUC and UCS-2 Code Set Considerations” on page 419.

---

## Chapter 20. Programming in REXX

Programming Considerations . . . . .	607	Supported SQL Data Types . . . . .	617
Language Restrictions . . . . .	608	Using Cursors in REXX . . . . .	619
Registering SQLEXEC, SQLDBS and		Execution Requirements for REXX . . . . .	619
SQLDB2. . . . .	608	Bind Files for REXX. . . . .	620
Embedding SQL Statements . . . . .	609	API Syntax. . . . .	621
Host Variables. . . . .	611	REXX Stored Procedures . . . . .	623
Naming Host Variables. . . . .	611	Calling Stored Procedures. . . . .	623
Referencing Host Variables . . . . .	611	Considerations on the Client . . . . .	624
Indicator Variables in REXX . . . . .	612	Considerations on the Server. . . . .	624
Predefined REXX Variables . . . . .	612	Retrieving Precision and SCALE	
LOB Host Variables in REXX. . . . .	614	Values from SQLDA Decimal Fields	625
LOB Locator Declarations in REXX. . . . .	615	Japanese or Traditional Chinese EUC	
LOB File Reference Declarations in REXX	615	Considerations . . . . .	625
Clearing LOB Host Variables. . . . .	617		

---

### Programming Considerations

Special host-language programming considerations are discussed in the following pages. Included is information on embedding SQL statements, language restrictions, and supported data types for host variables.

**Note:** REXX support stabilized in DB2 Version 5, and no enhancements for REXX support are planned for the future. For example, REXX cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 after Version 5, such as table names from 19 to 128 bytes long, you must not write your applications in REXX.

Because REXX is an interpreted language, no precompiler, compiler, or linker is used. Instead, three DB2 APIs are used to create DB2 applications in REXX. Use these APIs to access different elements of DB2.

#### SQLEXEC

Supports the SQL language

#### SQLDBS

Supports command-like versions of DB2 APIs.

#### SQLDB2

Supports a REXX specific interface to the command-line processor. See “API Syntax” on page 621 for details and restrictions on how this interface can be used.

---

## Language Restrictions

It is possible that tokens within statements or commands that are passed to the SQLEXEC, SQLDBS, and SQLDB2 routines could correspond to REXX variables. In this case, the REXX interpreter substitutes the variable's value before calling SQLEXEC, SQLDBS, or SQLDB2.

To avoid this situation, enclose statement strings in quotation marks ( ' ' or " "). If you do not use quotation marks, any conflicting variable names are resolved by the REXX interpreter, instead of being passed to the SQLEXEC, SQLDBS or SQLDB2 routines.

Compound SQL is not supported in REXX/SQL.

REXX/SQL stored procedures are supported on the OS/2 and Windows 32-bit operating systems, but not on AIX.

### Registering SQLEXEC, SQLDBS and SQLDB2

Before using any of the DB2 APIs or issuing SQL statements in an application, you must register the SQLDBS, SQLDB2 and SQLEXEC routines. This notifies the REXX interpreter of the REXX/SQL entry points. The method you use for registering varies slightly between the OS/2 and AIX platforms. The following examples show the correct syntax for registering each routine:

#### Sample registration on OS/2 or Windows

```
/* ----- Register SQLDBS with REXX ----- */
If Rxfuncquery('SQLDBS') <> 0 then
 rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
If rcy \= 0 then
 do
 say 'SQLDBS was not successfully added to the REXX environment'
 signal rxx_exit
 end

/* ----- Register SQLDB2 with REXX ----- */
If Rxfuncquery('SQLDB2') <> 0 then
 rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
If rcy \= 0 then
 do
 say 'SQLDB2 was not successfully added to the REXX environment'
 signal rxx_exit
 end

/* ----- Register SQLEXEC with REXX ----- */
If Rxfuncquery('SQLEXEC') <> 0 then
 rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
If rcy \= 0 then
```

```

do
 say 'SQLEXEC was not successfully added to the REXX environment'
 signal rxx_exit
end

```

### Sample registration on AIX

```

/* ----- Register SQLDBS, SQLDB2 and SQLEXEC with REXX -----*/
rcy = SysAddFuncPkg("db2rexx")
If rcy \= 0 then
do
 say 'db2rexx was not successfully added to the REXX environment'
 signal rxx_exit
end

```

On OS/2, the RxFuncAdd commands need to be executed only once for all sessions.

On AIX, the SysAddFuncPkg should be executed in every REXX/SQL application.

Details on the RXfuncadd and SysAddFuncPkg APIs are available in the REXX documentation for OS/2 and AIX, respectively.

---

## Embedding SQL Statements

Use the SQLEXEC routine to process all SQL statements. The character string arguments for the SQLEXEC routine are made up of the following elements:

- SQL keywords
- Pre-declared identifiers
- Statement host variables.

Make each request by passing a valid SQL statement to the SQLEXEC routine. Use the following syntax:

```
CALL SQLEXEC 'statement'
```

SQL statements can be continued onto more than one line. Each part of the statement should be enclosed in single quotation marks, and a comma must delimit additional statement text as follows:

```

CALL SQLEXEC 'SQL text',
 'additional text',
 .
 .
 .
 'final text'

```

The following is an example of embedding an SQL statement in REXX:

```

statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF (SQLCA.SQLCODE < 0) THEN
 SAY 'Update Error: SQLCODE = ' SQLCA.SQLCODE

```

In this example, the SQLCODE field of the SQLCA structure is checked to determine whether the update was successful.

The following rules apply to embedded SQL statements:

- The following SQL statements can be passed directly to the SQLEXEC routine:

```

CALL
CLOSE
COMMIT
CONNECT
CONNECT TO
CONNECT RESET
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FREE LOCATOR
OPEN
PREPARE
RELEASE
ROLLBACK
SET CONNECTION

```

Other SQL statements must be processed dynamically using the EXECUTE IMMEDIATE, or PREPARE and EXECUTE statements in conjunction with the SQLEXEC routine.

- You cannot use host variables in the CONNECT and SET CONNECTION statements in REXX.
- Cursor names and statement names are predefined as follows:

**c1 to c100**

Cursor names, which range from *c1* to *c50* for cursors declared without the WITH HOLD option, and *c51* to *c100* for cursors declared using the WITH HOLD option.

The cursor name identifier is used for DECLARE, OPEN, FETCH, and CLOSE statements. It identifies the cursor used in the SQL request.

#### **s1 to s100**

Statement names, which range from *s1* to *s100*.

The statement name identifier is used with the DECLARE, DESCRIBE, PREPARE, and EXECUTE statements.

The pre-declared identifiers must be used for cursor and statement names. Other names are not allowed.

- When declaring cursors, the cursor name and the statement name should correspond in the DECLARE statement. For example, if *c1* is used as a cursor name, *s1* must be used for the statement name.
- Do not use comments within an SQL statement.

---

## **Host Variables**

Host variables are REXX language variables that are referenced within SQL statements. They allow an application to pass input data to DB2 and receive output data from DB2. REXX applications do not need to declare host variables, except for LOB locators and LOB file reference variables. Host variable data types and sizes are determined at run time when the variables are referenced. Apply the following rules when naming and using host variables.

### **Naming Host Variables**

Any properly named REXX variable can be used as a host variable. A variable name can be up to 64 characters long. Do not end the name with a period. A host variable name can consist of alphabetic characters, numerics, and the characters @, \_, !, ., ?, and \$.

### **Referencing Host Variables**

The REXX interpreter examines every string without quotation marks in a procedure. If the string represents a variable in the current REXX variable pool, REXX replaces the string with the current value. The following is an example of how you can reference a host variable in REXX:

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

To ensure that a character string is not converted to a numeric data type, enclose the string with single quotation marks as in the following example:

```
VAR = '100'
```

REXX sets the variable *VAR* to the 3-byte character string 100. If single quotation marks are to be included as part of the string, follow this example:

```
VAR = "'100'"
```

When inserting numeric data into a CHARACTER field, the REXX interpreter treats numeric data as integer data, thus you must concatenate numeric strings explicitly and surround them with single quotation marks.

### Indicator Variables in REXX

An indicator variable data type in REXX is a number without a decimal point. Following is an example of an indicator variable in REXX using the INDICATOR keyword.

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF (cmind < 0)
 SAY 'Commission is NULL'
```

In the above example, *cmind* is examined for a negative value. If it is not negative, the application can use the returned value of *cm*. If it is negative, the fetched value is NULL and *cm* should not be used. The database manager does not change the value of the host variable in this case.

### Predefined REXX Variables

SQLEXEC, SQLDBS and SQLDB2 set predefined REXX variables as a result of certain operations. These variables are:

#### RESULT

Each operation sets this return code. Possible values are:

- n*        Where *n* is a positive value indicating the number of bytes in a formatted message. The GET ERROR MESSAGE API alone returns this value.
- 0**        The API was executed. The REXX variable SQLCA contains the completion status of the API. If SQLCA.SQLCODE is not zero, SQLMSG contains the text message associated with that value.
- 1**        There is not enough memory available to complete the API. The requested message was not returned.
- 2**        SQLCA.SQLCODE is set to 0. No message was returned.
- 3**        SQLCA.SQLCODE contained an invalid SQLCODE. No message was returned.
- 6**        The SQLCA REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason.



- 7 The SQLMSG REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason.
- 8 The SQLCA.SQLCODE REXX variable could not be fetched from the REXX variable pool.
- 9 The SQLCA.SQLCODE REXX variable was truncated during the fetch. The maximum length for this variable is 5 bytes.
- 10 The SQLCA.SQLCODE REXX variable could not be converted from ASCII to a valid long integer.
- 11 The SQLCA.SQLERRML REXX variable could not be fetched from the REXX variable pool.
- 12 The SQLCA.SQLERRML REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes.
- 13 The SQLCA.SQLERRML REXX variable could not be converted from ASCII to a valid short integer.
- 14 The SQLCA.SQLERRMC REXX variable could not be fetched from the REXX variable pool.
- 15 The SQLCA.SQLERRMC REXX variable was truncated during the fetch. The maximum length for this variable is 70 bytes.
- 16 The REXX variable specified for the error text could not be set.
- 17 The SQLCA.SQLSTATE REXX variable could not be fetched from the REXX variable pool.
- 18 The SQLCA.SQLSTATE REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes.

**Note:** The values -8 through -18 are returned only by the GET ERROR MESSAGE API.

### **SQLMSG**

If SQLCA.SQLCODE is not 0, this variable contains the text message associated with the error code.

### **SQLISL**

The isolation level. Possible values are:

- RR** Repeatable read.
- RS** Read stability.
- CS** Cursor stability. This is the default.
- UR** Uncommitted read.

**NC** No commit (NC is only supported by some host or AS/400 servers.)

### **SQLCA**

The SQLCA structure updated after SQL statements are processed and DB2 APIs are called. The entries of this structure are described in the *Administrative API Reference*.

### **SQLRODA**

The input/output SQLDA structure for stored procedures invoked using the CALL statement. It is also the output SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API. The entries of this structure are described in the *Administrative API Reference*.

### **SQLRIDA**

The input SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API. The entries of this structure are described in the *Administrative API Reference*.

### **SQLRDAT**

An SQLCHAR structure for server procedures invoked using the Database Application Remote Interface (DARI) API. The entries of this structure are described in the *Administrative API Reference*.

## **LOB Host Variables in REXX**

When you fetch a LOB column into a REXX host variable, it will be stored as a simple (that is, uncounted) string. This is handled in the same manner as all character-based SQL types (such as CHAR, VARCHAR, GRAPHIC, LONG, and so on). On input, if the size of the contents of your host variable is larger than 32K, or if it meets other criteria set out below, it will be assigned the appropriate LOB type.

In REXX SQL, LOB types are determined from the string content of your host variable as follows:

Host variable string content	Resulting LOB type
:hv1='ordinary quoted string longer than 32K ...'	CLOB
:hv2="string with embedded delimiting quotation marks ", "longer than 32K..."	CLOB
:hv3="G'DBCS string with embedded delimiting single ", "quotation marks, beginning with G, longer than 32K..."	DBCLOB
:hv4="BIN'string with embedded delimiting single ", "quotation marks, beginning with BIN, any length..."	BLOB

# LOB Locator Declarations in REXX

“Syntax for LOB Locator Host Variables in REXX” shows the syntax for declaring LOB locator host variables in REXX.

## Syntax for LOB Locator Host Variables in REXX



You must declare LOB locator host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as locators for the remainder of the program. Locator values are stored in REXX variables in an internal format.

### Example:

```
CALL SQLEEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

Data represented by LOB locators returned from the engine can be freed in REXX/SQL using the `FREE LOCATOR` statement which has the following format:

## Syntax for FREE LOCATOR Statement



### Example:

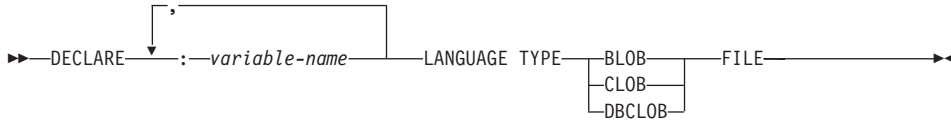
```
CALL SQLEEXEC 'FREE LOCATOR :hv1, :hv2'
```

# LOB File Reference Declarations in REXX

You must declare LOB file reference host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as LOB file references for the remainder of the program.

“Syntax for LOB File Reference Variables in REXX” shows the syntax for declaring LOB file reference host variables in REXX.

## REXX File Reference Declarations



Example:

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

File reference variables in REXX contain three fields. For the above example they are:

### hv3.FILE\_OPTIONS.

Set by the application to indicate how the file will be used.

### hv3.DATA\_LENGTH.

Set by DB2 to indicate the size of the file.

### hv3.NAME.

Set by the application to the name of the LOB file.

For FILE\_OPTIONS, the application sets the following keywords:

Keyword (Integer Value)	Meaning
-------------------------	---------

<b>READ (2)</b>	File is to be used for input. This is a regular file that can be opened, read and closed. The length of the data in the file (in bytes) is computed (by the application requestor code) upon opening the file.
<b>CREATE (8)</b>	On output, create a new file. If the file already exists, it is an error. The length (in bytes) of the file is returned in the DATA_LENGTH field of the file reference variable structure.
<b>OVERWRITE (16)</b>	On output, the existing file is overwritten if it exists, otherwise a new file is created. The length (in bytes) of the file is returned in the DATA_LENGTH field of the file reference variable structure.
<b>APPEND (32)</b>	The output is appended to the file if it exists, otherwise a new file is created. The length (in bytes) of the data that was added to the file (not the total file length) is returned in the DATA_LENGTH field of the file reference variable structure.

**Note:** A file reference host variable is a compound variable in REXX, thus you must set values for the NAME, NAME\_LENGTH and FILE\_OPTIONS fields in addition to declaring them.

## Clearing LOB Host Variables

On OS/2 it may be necessary to explicitly clear REXX SQL LOB locator and file reference host variable declarations as they remain in effect after your application program ends. This is because the application process does not exit until the session in which it is run is closed. If REXX SQL LOB declarations are not cleared, they may interfere with other applications that are running in the same session after a LOB application has been executed.

The syntax to clear the declaration is:

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

You should code this statement at the end of LOB applications. Note that you can code it anywhere as a precautionary measure to clear declarations which might have been left by previous applications (for example, at the beginning of a REXX SQL application).

---

## Supported SQL Data Types

Certain predefined REXX data types correspond to DB2 column types. Table 24 shows how SQLEXEC and SQLDBS interpret REXX variables in order to convert their contents to DB2 data types.

**Note:** There is no host variable support for the DATALINK data type in any of the DB2 host languages.

Table 24. SQL Column Types Mapped to REXX Declarations

SQL Column Type <sup>1</sup>	REXX Data Type	SQL Column Type Description
SMALLINT (500 or 501)	A number without a decimal point ranging from -32 768 to 32 767	16-bit signed integer
INTEGER (496 or 497)	A number without a decimal point ranging from -2 147 483 648 to 2 147 483 647	32-bit signed integer
REAL <sup>2</sup> (480 or 481)	A number in scientific notation ranging from -3.40282346 x 10 <sup>38</sup> to 3.40282346 x 10 <sup>38</sup>	Single-precision floating point
DOUBLE <sup>3</sup> (480 or 481)	A number in scientific notation ranging from -1.79769313 x 10 <sup>308</sup> to 1.79769313 x 10 <sup>308</sup>	Double-precision floating point
DECIMAL(p,s) (484 or 485)	A number with a decimal point	Packed decimal

Table 24. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type <sup>1</sup>	REXX Data Type	SQL Column Type Description
CHAR( <i>n</i> ) (452 or 453)	A string with a leading and trailing quote ('), which has length <i>n</i> after removing the two quote marks  A string of length <i>n</i> with any non-numeric characters, other than leading and trailing blanks or the E in scientific notation	Fixed-length character string of length <i>n</i> where <i>n</i> is from 1 to 254
VARCHAR( <i>n</i> ) (448 or 449)	Equivalent to CHAR( <i>n</i> )	Variable-length character string of length <i>n</i> , where <i>n</i> ranges from 1 to 4000
LONG VARCHAR (456 or 457)	Equivalent to CHAR( <i>n</i> )	Variable-length character string of length <i>n</i> , where <i>n</i> ranges from 1 to 32 700
CLOB( <i>n</i> ) (408 or 409)	Equivalent to CHAR( <i>n</i> )	Large object variable-length character string of length <i>n</i> , where <i>n</i> ranges from 1 to 2 147 483 647
CLOB locator variable <sup>4</sup> (964 or 965)	DECLARE : <i>var_name</i> LANGUAGE TYPE CLOB LOCATOR	Identifies CLOB entities residing on the server
CLOB file reference variable <sup>4</sup> (808 or 809)	DECLARE : <i>var_name</i> LANGUAGE TYPE CLOB FILE	Descriptor for file containing CLOB data
BLOB( <i>n</i> ) (404 or 405)	A string with a leading and trailing apostrophe, preceded by BIN, containing <i>n</i> characters after removing the preceding BIN and the two apostrophes.	Large object variable-length binary string of length <i>n</i> , where <i>n</i> ranges from 1 to 2 147 483 647
BLOB locator variable <sup>4</sup> (960 or 961)	DECLARE : <i>var_name</i> LANGUAGE TYPE BLOB LOCATOR	Identifies BLOB entities on the server
BLOB file reference variable <sup>4</sup> (804 or 805)	DECLARE : <i>var_name</i> LANGUAGE TYPE BLOB FILE	Descriptor for the file containing BLOB data
DATE (384 or 385)	Equivalent to CHAR(10)	10-byte character string
TIME (388 or 389)	Equivalent to CHAR(8)	8-byte character string
TIMESTAMP (392 or 393)	Equivalent to CHAR(26)	26-byte character string
<b>Note:</b> The following data types are only available in the DBCS environment.		
GRAPHIC( <i>n</i> ) (468 or 469)	A string with a leading and trailing apostrophe preceded by a G or N, containing <i>n</i> DBCS characters after removing the preceding character and the two apostrophes	Fixed-length graphic string of length <i>n</i> , where <i>n</i> is from 1 to 127
VARGRAPHIC( <i>n</i> ) (464 or 465)	Equivalent to GRAPHIC( <i>n</i> )	Variable-length graphic string of length <i>n</i> , where <i>n</i> ranges from 1 to 2000

Table 24. SQL Column Types Mapped to REXX Declarations (continued)

SQL Column Type <sup>1</sup>	REXX Data Type	SQL Column Type Description
LONG VARCHAR(n) (472 or 473)	Equivalent to GRAPHIC(n)	Long variable-length graphic string of length <i>n</i> , where <i>n</i> ranges from 1 to 16 350
DBCLOB(n) (412 or 413)	Equivalent to GRAPHIC(n)	Large object variable-length graphic string of length <i>n</i> , where <i>n</i> ranges from 1 to 1 073 741 823
DBCLOB locator variable <sup>4</sup> (968 or 969)	DECLARE <i>.var_name</i> LANGUAGE TYPE DBCLOB LOCATOR	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable <sup>4</sup> (812 or 813)	DECLARE <i>.var_name</i> LANGUAGE TYPE DBCLOB FILE	Descriptor for file containing DBCLOB data

**Note:**

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string.
2. FLOAT(*n*) where  $0 < n < 25$  is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
  - FLOAT
  - FLOAT(*n*) where  $24 < n < 54$  is
  - DOUBLE PRECISION
4. This is not a column type but a host variable type.

## Using Cursors in REXX

When a cursor is declared in REXX, the cursor is associated with a query. The query is associated with a statement name assigned in the PREPARE statement. Any referenced host variables are represented by parameter markers. The following example shows a DECLARE statement associated with a dynamic SELECT statement.

```
prep_string = "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"
CALL SQLEXEC 'PREPARE S1 FROM :prep_string';
CALL SQLEXEC 'DECLARE C1 CURSOR FOR S1';
CALL SQLEXEC 'OPEN C1 USING :schema_name';
```

## Execution Requirements for REXX

REXX applications are not precompiled, compiled, or linked.

On OS/2, your application file must have a .CMD extension. After creation, you can run your application directly from the operating system command prompt.

On Windows 32-bit operating systems, your application file can have any name. After creation, you can run your application from the operating system command prompt by invoking the REXX interpreter as follows:

```
REXX file_name
```

On AIX, your application file can have any extension. You can run your application using either of the following two methods:

1. At the shell command prompt, type rexx name where name is the name of your REXX program.
2. If the first line of your REXX program contains a "magic number" (#!) and identifies the directory where the REXX/6000 interpreter resides, you can run your REXX program by typing its name at the shell command prompt. For example, if the REXX/6000 interpreter file is in the /usr/bin directory, include the following as the very first line of your REXX program:

```
#!/usr/bin/rexx
```

Then, make the program executable by typing the following command at the shell command prompt:

```
chmod +x name
```

Run your REXX program by typing its file name at the shell command prompt.

**Note:** On AIX, you should set the LIBPATH environment variable to include the directory where the REXX SQL library, db2rexx is located. For example:

```
export LIBPATH=/lib:/usr/lib:/usr/lpp/db2_05_00/lib
```

## Bind Files for REXX

Five bind files are provided to support REXX applications. The names of these files are included in the DB2UBIND.LST file. Each bind file was precompiled using a different isolation level; therefore, there are five different packages stored in the database.

The five bind files are:

### **DB2ARXCS.BND**

Supports the cursor stability isolation level

### **DB2ARXRR.BND**

Supports the repeatable read isolation level

### **DB2ARXUR.BND**

Supports the uncommitted read isolation level



**DB2ARXRS.BND**

Supports the read stability isolation level.

**DB2ARXNC.BND**

Supports the no commit isolation level. This isolation level is used when working with some host or AS/400 database servers. On other databases, it behaves like the uncommitted read isolation level.

**Note:** In some cases, it may be necessary to explicitly bind these files to the database.

When you use the SQLEXEC routine, the package created with cursor stability is used as a default. If you require one of the other isolation levels, you can change isolation levels with the SQLDBS CHANGE SQL ISOLATION LEVEL API, before connecting to the database. This will cause subsequent calls to the SQLEXEC routine to be associated with the specified isolation level.

OS/2 REXX applications cannot assume that the default isolation level is in effect unless they know that no other REXX programs in the session have changed the setting. Before connecting to a database, a REXX application should explicitly set the isolation level.

---

**API Syntax**

Use the SQLDBS routine to call DB2 APIs with the following syntax:

```
CALL SQLDBS 'command string'
```

For information on how the DB2 APIs work, see the complete descriptions in the DB2 API chapter of the *Administrative API Reference*.

If a DB2 API you want to use cannot be called using the SQLDBS routine (and consequently, not listed in the *Administrative API Reference*), you may still call the API by calling the DB2 command line processor (CLP) from within the REXX application. However, since the DB2 CLP directs output either to the standard output device or to a specified file, your REXX application cannot directly access the output from the called DB2 API nor can it easily make a determination as to whether the called API is successful or not. The SQLDB2 API provides an interface to the DB2 CLP that provides direct feedback to your REXX application on the success or failure of each called API by setting the compound REXX variable, SQLCA, after each call.

You can use the SQLDB2 routine to call DB2 APIs using the following syntax:

```
CALL SQLDB2 'command string'
```

where 'command string' is a string that can be processed by the command-line processor (CLP). Refer to the *Command Reference* for the syntax of strings that can be processed by the CLP.

Calling a DB2 API using SQLDB2 is equivalent to calling the CLP directly, except for the following:

- The call to the CLP executable is replaced by the call to SQLDB2 (all other CLP options and parameters are specified the same way).
- The REXX compound variable SQLCA is set after calling the SQLDB2 but is not set after calling the CLP executable.
- The default display output of the CLP is set to off when you call SQLDB2, whereas the display is set to on output when you call the CLP executable. Note that you can turn the display output of the CLP to on by passing the +o or the -o- option to the SQLDB2.

Since the only REXX variable that is set after you call SQLDB2 is the SQLCA, you only use this routine to call DB2 APIs that do not return any data other than the SQLCA and that are not currently implemented through the SQLDBS interface. Thus, only the following DB2 APIs are supported by SQLDB2:

Activate Database

Add Node

Bind for DB2 Version 1<sup>(1)</sup> <sup>(2)</sup>

Bind for DB2 Version 2 or 5<sup>(1)</sup>

Create Database at Node

Drop Database at Node

Drop Node Verify

Deactivate Database

Deregister

Load<sup>(3)</sup>

Load Query

Precompile Program<sup>(1)</sup>

Rebind Package<sup>(1)</sup>

Redistribute Nodegroup

Register

Start Database Manager

Stop Database Manager

### Notes on DB2 APIs Supported by SQLDB2:

1. These commands require a CONNECT statement through the SQLDB2 interface. Connections using the SQLDB2 interface are not accessible to the SQLEXEC interface and connections using the SQLEXEC interface are not accessible to the SQLDB2 interface.
2. Is supported on OS/2 through the SQLDB2 interface.
3. The optional output parameter, pLoadInfoOut for the Load API is not returned to the application in REXX. Refer to the *Administrative API Reference* for more information on the Load API and its parameters.

**Note:** Although the SQLDB2 routine is intended to be used only for the DB2 APIs listed above, it can also be used for other DB2 APIs that are not supported through the SQLDBS routine. Alternatively, the DB2 APIs can be accessed through the CLP from within the REXX application.

---

## REXX Stored Procedures

REXX SQL applications can call stored procedures at the database server by using the SQL CALL statement. The stored procedure can be written in any language supported on that server, except for REXX on AIX systems. (Client applications may be written in REXX on AIX systems, but, as with other languages, they cannot call a stored procedure written in REXX on AIX.)

### Calling Stored Procedures

The CALL statement allows a client application to pass data to, and receive data from, a server stored procedure. The interface for both input and output data is a list of host variables (refer to the *SQL Reference* for details). Because REXX generally determines the type and size of host variables based on their content, any output-only variables passed to CALL should be initialized with *dummy* data similar in type and size to the expected output.

Data can also be passed to stored procedures through SQLDA REXX variables, using the USING DESCRIPTOR syntax of the CALL statement. Table 25 shows how the SQLDA is set up. In the table, 'value' is the stem of a REXX host variable that contains the values needed for the application. For the DESCRIPTOR, 'n' is a numeric value indicating a specific *sqlvar* element of the SQLDA. The numbers on the right refer to the notes following Table 25.

Table 25. Client-side REXX SQLDA for Stored Procedures using the CALL Statement

USING DESCRIPTOR	:value.SQLD	1
	:value.n.SQTYPE	1
	:value.n.SQLEN	1

Table 25. Client-side REXX SQLDA for Stored Procedures using the CALL Statement (continued)

:value.n.SQLDATA	1	2
:value.n.SQLDIND	1	2

**Notes:**

1. Before invoking the stored procedure, the client application must initialize the REXX variable with appropriate data.

When the SQL CALL statement is executed, the database manager allocates storage and retrieves the value of the REXX variable from the REXX variable pool. For an SQLDA used in a CALL statement, the database manager allocates storage for the SQLDATA and SQLIND fields based on the SQLTYPE and SQLLEN values.

In the case of a REXX stored procedure (that is, the procedure being called is itself written in OS/2 REXX), the data passed by the client from either type of CALL statement or the DARI API is placed in the REXX variable pool at the database server using the following predefined names:

**SQLRIDA**

Predefined name for the REXX input SQLDA variable

**SQLRODA**

Predefined name for the REXX output SQLDA variable

2. When the stored procedure terminates, the database manager also retrieves the value of the variables from the stored procedure. The values are returned to the client application and placed in the client's REXX variable pool.

**Considerations on the Client**

When using host variables in the CALL statement, initialize each host variable to a value that is type compatible with any data that is returned to the host variable from the server procedure. You should perform this initialization even if the corresponding indicator is negative.

When using descriptors, SQLDATA must be initialized and contain data that is type compatible with any data that is returned from the server procedure. You should perform this initialization even if the SQLIND field contains a negative value.

**Considerations on the Server**

Ensure that all the SQLDATA fields and SQLIND (if it is a nullable type) of the predefined output sqlda SQLRODA are initialized. For example, if

SQLRODA.SQLD is 2, the following fields must contain some data (even if the corresponding indicators are negative and the data is not passed back to the client):

- SQLRODA.1.SQLDATA
- SQLRODA.2.SQLDATA

### Retrieving Precision and SCALE Values from SQLDA Decimal Fields

To retrieve the precision and scale values for decimal fields from the SQLDA structure returned by the database manager, use the `sqlllen.scale` and `sqlllen.precision` values when you initialize the SQLDA output in your REXX program. For example:

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485 /* DECIMAL DATA TYPE */
io_sqlda.1.sqlllen.scale = 2 /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqlllen.precision = 7 /* WIDTH OF DECIMAL */
io_sqlda.1.sqldata = 00000.00 /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1 /* NO INPUT DATA */
.
.
.
```

---

## Japanese or Traditional Chinese EUC Considerations

REXX applications are not supported under Japanese or Traditional Chinese EUC environments.



---

## Part 7. Appendixes





## Appendix A. Supported SQL Statements (DB2 Universal Database)

Table 26:

- Lists all the supported SQL statements
- Indicates (with an 'X') if they can be executed dynamically
- Indicates (with an 'X') if they are supported by the command line processor (CLP)
- Indicates (with an 'X' or DB2 CLI function name) if the statement can be executed using the DB2 Call Level Interface (DB2 CLI).

You can use Table 26 as a quick reference aid. For a complete discussion of all the statements, including their syntax, refer to the *SQL Reference*.

Table 26. SQL Statements (DB2 Universal Database)

SQL Statement	Dynamic <sup>1</sup>	Command Line Processor (CLP)	Call Level Interface <sup>3</sup> (CLI)
ALTER { BUFFERPOOL, NODEGROUP, TABLE, TABLESPACE, TYPE, VIEW }	X	X	X
BEGIN DECLARE SECTION <sup>2</sup>			
CALL		X <sup>9</sup>	X <sup>4</sup>
CLOSE		X	SQLCloseCursor(), SQLFreeStmt()
COMMENT ON	X	X	X
COMMIT	X	X	SQLEndTran, SQLTransact()
Compound SQL			X <sup>4</sup>
CONNECT (Type 1)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()
CONNECT (Type 2)		X	SQLBrowseConnect(), SQLConnect(), SQLDriverConnect()

Table 26. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic <sup>1</sup>	Command Line Processor (CLP)	Call Level Interface <sup>3</sup> (CLI)
CREATE { ALIAS, BUFFERPOOL, DISTINCT TYPE, EVENT MONITOR, FUNCTION, INDEX, NODEGROUP, PROCEDURE, SCHEMA, TABLE, TABLESPACE, TRIGGER, TYPE, VIEW }	X	X	X
DECLARE CURSOR <sup>2</sup>		X	SQLAllocStmt()
DELETE	X	X	X
DESCRIBE <sup>8</sup>		X	SQLColAttributes(), SQLDescribeCol(), SQLDescribeParam() <sup>6</sup>
DISCONNECT		X	SQLDisconnect()
DROP	X	X	X
END DECLARE SECTION <sup>2</sup>			
EXECUTE			SQLExecute()
EXECUTE IMMEDIATE			SQLExecDirect()
EXPLAIN	X	X	X
FETCH		X	SQLExtendedFetch() <sup>7</sup> , SQLFetch(), SQLFetchScroll() <sup>7</sup>
FREE LOCATOR			X <sup>4</sup>
GRANT	X	X	X
INCLUDE <sup>2</sup>			
INSERT	X	X	X
LOCK TABLE	X	X	X
OPEN		X	SQLExecute(), SQLExecDirect()
PREPARE			SQLPrepare()
RELEASE		X	
RENAME TABLE	X	X	X
REVOKE	X	X	X
ROLLBACK	X	X	SQLEndTran(), SQLTransact()
select-statement	X	X	X
SELECT INTO			

Table 26. SQL Statements (DB2 Universal Database) (continued)

SQL Statement	Dynamic <sup>1</sup>	Command Line Processor (CLP)	Call Level Interface <sup>3</sup> (CLI)
SET CONNECTION		X	SQLSetConnection()
SET CONSTRAINTS	X	X	X
SET CURRENT DEGREE	X	X	X
SET CURRENT EXPLAIN MODE	X	X	X, SQLSetConnectAttr()
SET CURRENT EXPLAIN SNAPSHOT	X	X	X, SQLSetConnectAttr()
SET CURRENT FUNCTION PATH	X	X	X
SET CURRENT PACKAGESET			
SET CURRENT QUERY OPTIMIZATION	X	X	X
SET EVENT MONITOR STATE	X	X	X
SET transition-variable <sup>5</sup>	X	X	X
SIGNAL SQLSTATE <sup>5</sup>	X	X	X
UPDATE	X	X	X
VALUES INTO			
WHENEVER <sup>2</sup>			

**Note:**

1. You can code all statements in this list as static SQL, but only those marked with X as dynamic SQL.
2. You cannot execute this statement.
3. An X indicates that you can execute this statement using either `SQLExecDirect()` or `SQLPrepare()` and `SQLExecute()`. If there is an equivalent DB2 CLI function, the function name is listed.
4. Although this statement is not dynamic, with DB2 CLI you can specify this statement when calling either `SQLExecDirect()`, or `SQLPrepare()` and `SQLExecute()`.
5. You can only use this within CREATE TRIGGER statements.
6. You can only use the SQL DESCRIBE statement to describe output, whereas with DB2 CLI you can also describe input (using the `SQLDescribeParam()` function).
7. You can only use the SQL FETCH statement to fetch one row at a time in one direction, whereas with the DB2 CLI `SQLExtendedFetch()` and `SQLFetchScroll()` functions, you can fetch into arrays. Furthermore, you can fetch in any direction, and at any position in the result set.
8. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command. For information on the DESCRIBE SQL statement, refer to the *SQL Reference*. For information on the DESCRIBE CLP command, refer to the *Command Reference*.
9. When CALL is issued through the command line processor, only certain procedures and their respective parameters are supported (see “Installing, Replacing, and Removing JAR Files” on page 563).



---

## Appendix B. Sample Programs

This section provides information on the sample programs supplied with DB2. All sample programs can be found in the `samples` subdirectory of the `sqllib` directory. There is a subdirectory for each supported language.

The sample programs used in this book show examples of embedded SQL statements and API calls in the supported host languages. The sample programs are written to be short and simple. Production applications should check the return codes, and especially the `SQLCODE` or `SQLSTATE` from all API calls and SQL statements. For information on handling error conditions, `SQLCODEs`, and `SQLSTATEs`, see “Diagnostic Handling and the SQLCA Structure” on page 117. See the *Application Building Guide* for details on how to install, build, and execute these programs in your environment.

### Notes:

1. This section describes sample programs for the programming languages for all platforms supported by DB2. Not all sample programs have been ported to all supported programming languages.
2. DB2 sample programs are provided “as is” without any warranty of any kind. The user, and not IBM, assumes the entire risk of quality, performance, and repair of any defects.

The sample programs come with the DB2 SDK. You can use the sample programs as templates to create your own applications. The file extensions for each supported language, as well as for programs categorized by group, are given in the following tables:

### Sample File Extensions by Language

Table 27 on page 635.

### Sample File Extensions by Program Group

Table 28 on page 635.

The following tables document the sample programs by type:

### DB2 API Sample Programs with No Embedded SQL

Table 29 on page 638.

### DB2 API Embedded SQL Sample Programs

Table 30 on page 640.

### Embedded SQL Sample Programs with No DB2 APIs

Table 31 on page 642.

**User-Defined Function Sample Programs**

Table 32 on page 645

**DB2 CLI Sample Programs**

Table 33 on page 645.

**Java JDBC Sample Programs**

Table 34 on page 648.

**Java SQLJ Sample Programs**

Table 35 on page 650.

**ActiveX Data Objects, Remote Data Objects, and Microsoft Transaction Server Sample Programs**

Table 36 on page 650.

**Object Linking and Embedding (OLE) Automation Sample Programs**

Table 37 on page 651.

**Object Linking and Embedding Database (OLE DB) Table Functions**

Table 38 on page 652.

**Command Line Processor (CLP) Sample Programs**

Table 39 on page 652.

**Log Management User Exit Programs**

Table 40 on page 653.

**Notes:**

1. Table 30 on page 640 contains programs that have both DB2 APIs and embedded SQL statements. For all DB2 API sample programs, please see both Table 29 on page 638 and Table 30 on page 640. For all embedded SQL sample programs (except for Java SQLJ), please see both Table 30 on page 640 and Table 31 on page 642.
2. Table 32 on page 645 of UDF sample programs does not contain DB2 CLI UDF programs. For these, please see Table 33 on page 645.

Table 27. Sample File Extensions by Language

Language	Directory	Embedded SQL Programs	Non-embedded SQL Programs
C	samples/c samples/cli (CLI programs)	.sqc	.c
C++	samples/cpp	.sqc (UNIX) .sqx (Windows & OS/2)	.C (UNIX) .cxx (Windows & OS/2)
COBOL	samples/cobol samples/cobol_mf	.sqb	.cbl
Fortran	samples/fortran	.sqf	.f (UNIX) .for (OS/2)
JAVA	samples/java	.sqlj	.java
REXX	samples/rexx	.cmd	.cmd

Table 28. Sample File Extensions by Program Group

Sample Group	Directory	File Extension
CLP	samples/clp	.db2
OLE	samples\ole\msvb (Visual Basic) samples\ole\msvc (Visual C++)	.bas .vbp (Visual Basic) .cpp (Visual C++)
OLE DB	samples\oledb	.db2
ADO, RDO, MTS	samples\ADO\VB (Visual Basic) samples\ADO\VC (Visual C++) samples\RDO samples\MTS	.bas .frm .vbp (Visual Basic) .cpp .dsp .dsw (Visual C++)
User Exit	samples/c	.cad (OS/2) .cadsm (UNIX & Windows) .cdisk (UNIX & Windows) .ctape (UNIX)

**Note:****Directory Delimiters**

On UNIX are /. On OS/2 and Windows platforms, are \. In the tables, the UNIX delimiters are used unless the directory is only available on Windows and/or OS/2.

**File Extensions**

Are provided for the samples in the tables where only one extension exists.

**Embedded SQL Programs**

Require precompilation, except for REXX embedded SQL programs where the embedded SQL statements are interpreted when the program is run.

**IBM COBOL samples**

Are only supplied for the OS/2, AIX, and Windows 32-bit operating systems in the `cobol` subdirectory.

**Micro Focus Cobol Samples**

Are supplied on all platforms except Linux and Silicon Graphics IRIX. On all other platforms, the Micro Focus COBOL samples are in the `cobol_mf` subdirectory.

**Fortran Samples**

Are only supplied on the AIX, HP-UX, Silicon Graphics IRIX, Solaris, and OS/2 platforms.

**Java Samples**

Are Java Database Connectivity (JDBC) applets, applications, and stored procedures, embedded SQL for Java (SQLJ) applets, applications, and stored procedures, as well as Java UDFs. Java samples are available on all supported DB2 platforms.

**REXX Samples**

Are only supplied for the AIX, OS/2, and Windows NT operating systems.

**CLP Samples**

Are Command Line Processor scripts that execute SQL statements.

**OLE Samples**

Are for Object Linking and Embedding (OLE) in Microsoft Visual Basic and Microsoft Visual C++, supplied for Windows 32-bit operating systems only.

**ADO, RDO, and MTS Samples**

Are ActiveX Data Objects samples in Microsoft Visual Basic and Microsoft Visual C++, and Remote Data Objects and Microsoft Transaction Server samples in Microsoft Visual Basic, supplied for Windows 32-bit operating systems only.

**User Exit samples**

Are Log Management User Exit programs used to archive and retrieve database log files. The files must be renamed with a `.c` extension and compiled as C language programs.

You can find the sample programs in the `samples` subdirectory of the directory where DB2 has been installed. There is a subdirectory for each supported language. The following examples show you how to locate the samples written in C or C++ on each supported platform.

- On UNIX platforms.



You can find the C source code for embedded SQL and DB2 API programs in `sqllib/samples/c` under your database instance directory; the C source code for DB2 CLI programs is in `sqllib/samples/cli`. For additional information about the programs in the samples tables, refer to the README file in the appropriate samples subdirectory under your DB2 instance. The README file will contain any additional samples that are not listed in this book.

- On OS/2 and Windows 32-bit operating systems.

You can find the C source code for embedded SQL and DB2 API programs in `%DB2PATH%\samples\c` under the DB2 install directory; the C source code for DB2 CLI programs is in `%DB2PATH%\samples\cli`. The variable `%DB2PATH%` determines where DB2 is installed. Depending on the drive where DB2 is installed, `%DB2PATH%` will point to `drive:\sqllib`. For additional information about the sample programs in the samples tables, refer to the README file in the appropriate `%DB2PATH%\samples` subdirectory. The README file will contain any additional samples that are not listed in this book.

If your platform is not addressed in Table 27 on page 635, please refer to the *Application Building Guide* for information specific to your environment.

The sample programs directory is typically read-only on most platforms. Before you alter or build the sample programs, copy them to your working directory.

**Note:** The sample programs that are shipped with DB2 Universal Database have dependencies on the English version of the sample database and the associated table and column names. If the sample database has been translated into another national language on your version of DB2 Universal Database, you need to update the name of the sample database, and the names of the tables and the columns coded in the supplied sample programs, to the names used in the translated sample database. Otherwise, you will experience problems running the sample programs as shipped.

Currently, the sample database is translated into the following languages:

- Brazilian Portuguese
- French
- Korean
- Norwegian
- Simplified Chinese

## DB2 API Non-Embedded SQL Samples

Table 29. DB2 API Sample Programs with No Embedded SQL

Sample Program	Included APIs
backrest	<ul style="list-style-type: none"><li>• sqlbftcq - Fetch Tablespace Container Query</li><li>• sqlbstsc - Set Tablespace Containers</li><li>• sqlfudb - Update Database Configuration</li><li>• sqlubkp - Backup Database</li><li>• sqluroll - Rollforward Database</li><li>• sqlurst - Restore Database</li></ul>
checkerr	<ul style="list-style-type: none"><li>• sqlaintp - Get Error Message</li><li>• sqllogstt - Get SQLSTATE Message</li></ul>
cli_info	<ul style="list-style-type: none"><li>• sqleqryi - Query Client Information</li><li>• sqleseti - Set Client Information</li></ul>
client	<ul style="list-style-type: none"><li>• sqleqryc - Query Client</li><li>• sqlesetc - Set Client</li></ul>
d_dbconf	<ul style="list-style-type: none"><li>• sqlentin - Attach</li><li>• sqledtin - Detach</li><li>• sqlfddb - Get Database Configuration Defaults</li></ul>
d_dbmcon	<ul style="list-style-type: none"><li>• sqlentin - Attach</li><li>• sqledtin - Detach</li><li>• sqlfdsys - Get Database Manager Configuration Defaults</li></ul>
db_udcs	<ul style="list-style-type: none"><li>• sqlentin - Attach</li><li>• sqlecrea - Create Database</li><li>• sqledrpd - Drop Database</li></ul>
db2mon	<ul style="list-style-type: none"><li>• sqlentin - Attach</li><li>• sqlmon - Get/Update Monitor Switches</li><li>• sqlmonss - Get Snapshot</li><li>• sqlmonsz - Estimate Size Required for sqlmonss() Output Buffer</li><li>• sqlmrset - Reset Monitor</li></ul>
dbcat	<ul style="list-style-type: none"><li>• sqlecadb - Catalog Database</li><li>• sqledcls - Close Database Directory Scan</li><li>• sqledgne - Get Next Database Directory Entry</li><li>• sqledosd - Open Database Directory Scan</li><li>• sqleuncd - Uncatalog Database</li></ul>

Table 29. DB2 API Sample Programs with No Embedded SQL (continued)

Sample Program	Included APIs
dbcmnt	<ul style="list-style-type: none"> <li>• sqledcgd - Change Database Comment</li> <li>• sqledcls - Close Database Directory Scan</li> <li>• sqledgne - Get Next Database Directory Entry</li> <li>• sqledosd - Open Database Directory Scan</li> <li>• sqleisig - Install Signal Handler</li> </ul>
dbconf	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlecrea - Create Database</li> <li>• sqledrpd - Drop Database</li> <li>• sqlfrdb - Reset Database Configuration</li> <li>• sqlfudb - Update Database Configuration</li> <li>• sqlfxdb - Get Database Configuration</li> </ul>
dbinst	<ul style="list-style-type: none"> <li>• sqleatcp - Attach and Change Password</li> <li>• sqleatin - Attach</li> <li>• sqledtin - Detach</li> <li>• sqlegins - Get Instance</li> </ul>
dbmconf	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqledtin - Detach</li> <li>• sqlfrsys - Reset Database Manager Configuration</li> <li>• sqlfusys - Update Database Manager Configuration</li> <li>• sqlfxsys - Get Database Manager Configuration</li> </ul>
dbsnap	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlmonss - Get Snapshot</li> </ul>
dbstart	<ul style="list-style-type: none"> <li>• sqlepstart - Start Database Manager</li> </ul>
dbstop	<ul style="list-style-type: none"> <li>• sqlefrce - Force Application</li> <li>• sqlepstp - Stop Database Manager</li> </ul>
dcscat	<ul style="list-style-type: none"> <li>• sqlegdad - Catalog DCS Database</li> <li>• sqlegdcl - Close DCS Directory Scan</li> <li>• sqlegdel - Uncatalog DCS Database</li> <li>• sqlegdge - Get DCS Directory Entry for Database</li> <li>• sqlegdgt - Get DCS Directory Entries</li> <li>• sqlegdsc - Open DCS Directory Scan</li> </ul>
dmscont	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlecrea - Create Database</li> <li>• sqledrpd - Drop Database</li> </ul>

Table 29. DB2 API Sample Programs with No Embedded SQL (continued)

Sample Program	Included APIs
ebscdicdb	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlcrea - Create Database</li> <li>• sqledrpd - Drop Database</li> </ul>
migrate	<ul style="list-style-type: none"> <li>• sqlemgdb - Migrate Database</li> </ul>
monreset	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlmrset - Reset Monitor</li> </ul>
monsz	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlmonss - Get Snapshot</li> <li>• sqlmonsz - Estimate Size Required for sqlmonss() Output Buffer</li> </ul>
nodecat	<ul style="list-style-type: none"> <li>• sqlectnd - Catalog Node</li> <li>• sqlencls - Close Node Directory Scan</li> <li>• sqlengne - Get Next Node Directory Entry</li> <li>• sqlenops - Open Node Directory Scan</li> <li>• sqleuncn - Uncatalog Node</li> </ul>
regder	<ul style="list-style-type: none"> <li>• sqledreg - Deregister</li> <li>• sqlregs - Register</li> </ul>
restart	<ul style="list-style-type: none"> <li>• sqlerstd - Restart Database</li> </ul>
setact	<ul style="list-style-type: none"> <li>• sqlsact - Set Accounting String</li> </ul>
setrundg	<ul style="list-style-type: none"> <li>• sqlsdeg - Set Runtime Degree</li> </ul>
sws	<ul style="list-style-type: none"> <li>• sqleatin - Attach</li> <li>• sqlmon - Get/Update Monitor Switches</li> </ul>
util	<ul style="list-style-type: none"> <li>• sqlaintp - Get Error Message</li> <li>• sqlgstt - Get SQLSTATE Message</li> </ul>

## DB2 API Embedded SQL Samples

Table 30. DB2 API Embedded SQL Sample Programs

Sample Program	Included APIs
asynrlog	<ul style="list-style-type: none"> <li>• sqlurlog - Asynchronous Read Log</li> </ul>
bindfile	<ul style="list-style-type: none"> <li>• sqlabndx - Bind</li> </ul>
dbauth	<ul style="list-style-type: none"> <li>• sqluadai - Get Authorizations</li> </ul>

Table 30. DB2 API Embedded SQL Sample Programs (continued)

Sample Program	Included APIs
dbstat	<ul style="list-style-type: none"> <li>• sqlureot - Reorganize Table</li> <li>• sqlustat - Runstats</li> </ul>
expsamp	<ul style="list-style-type: none"> <li>• sqluexpr - Export</li> <li>• sqluimpr - Import</li> </ul>
impexp	<ul style="list-style-type: none"> <li>• sqluexpr - Export</li> <li>• sqluimpr - Import</li> </ul>
loadqry	<ul style="list-style-type: none"> <li>• db2LoadQuery - Load Query</li> </ul>
makeapi	<ul style="list-style-type: none"> <li>• sqlabndx - Bind</li> <li>• sqlaprep - Precompile Program</li> <li>• sqlpstp - Stop Database Manager</li> <li>• sqlpstr - Start Database Manager</li> </ul>
qload	<ul style="list-style-type: none"> <li>• sqluqry - Load Query</li> </ul>
rebind	<ul style="list-style-type: none"> <li>• sqlarbnd - Rebind</li> </ul>
rechist	<ul style="list-style-type: none"> <li>• sqlubkp - Backup Database</li> <li>• sqluhcls - Close Recovery History File Scan</li> <li>• sqluhgne - Get Next Recovery History File Entry</li> <li>• sqluhops - Open Recovery History File Scan</li> <li>• sqluhprn - Prune Recovery History File</li> <li>• sqluhupd - Update Recovery History File</li> </ul>
tabscont	<ul style="list-style-type: none"> <li>• sqlbctcq - Close Tablespace Container Query</li> <li>• sqlbftcq - Fetch Tablespace Container Query</li> <li>• sqlbotcq - Open Tablespace Container Query</li> <li>• sqlbtcq - Tablespace Container Query</li> <li>• sqlefmem - Free Memory</li> </ul>
tabspace	<ul style="list-style-type: none"> <li>• sqlbctsq - Close Tablespace Query</li> <li>• sqlbftpq - Fetch Tablespace Query</li> <li>• sqlbgtss - Get Tablespace Statistics</li> <li>• sqlbmtsq - Tablespace Query</li> <li>• sqlbotsq - Open Tablespace Query</li> <li>• sqlbstpq - Single Tablespace Query</li> <li>• sqlefmem - Free Memory</li> </ul>

Table 30. DB2 API Embedded SQL Sample Programs (continued)

Sample Program	Included APIs
tload	<ul style="list-style-type: none"> <li>• sqluexpr - Export</li> <li>• sqluload - Load</li> <li>• sqluvqdp - Quiesce Tablespaces for Table</li> </ul>
tspac	<ul style="list-style-type: none"> <li>• sqlbctcq - Close Tablespace Container Query</li> <li>• sqlbctsq - Close Tablespace Query</li> <li>• sqlbftcq - Fetch Tablespace Container Query</li> <li>• sqlbftpq - Fetch Tablespace Query</li> <li>• sqlbgtss - Get Tablespace Statistics</li> <li>• sqlbmtsq - Tablespace Query</li> <li>• sqlbotcq - Open Tablespace Container Query</li> <li>• sqlbotsq - Open Tablespace Query</li> <li>• sqlbstpq - Single Tablespace Query</li> <li>• sqlbstsc - Set Tablespace Containers</li> <li>• sqlbtcq - Tablespace Container Query</li> <li>• sqlfmem - Free Memory</li> </ul>

## Embedded SQL Samples With No DB2 APIs

Table 31. Embedded SQL Sample programs with No DB2 APIs

Sample Program Name	Program Description
adhoc	Demonstrates dynamic SQL and the SQLDA structure to process SQL commands interactively. SQL commands are input by the user, and output corresponding to the SQL command is returned. See “Example: ADHOC Program” on page 159 for details.
advsql	Demonstrates the use of advanced SQL expressions like CASE, CAST, and scalar full selects.
blobfile	Demonstrates the manipulation of a Binary Large Object (BLOB), by reading a BLOB value from the sample database and placing it in a file. The contents of this file can be displayed using an external viewer.
calludf	Demonstrates calling user-defined functions (UDFs) created by the udf program, and stored on the server to access tables in the sample database.
columns	Demonstrates the use of a cursor that is processed using dynamic SQL. This program lists all the entries in the system table, SYSIBM.SYSTABLES, under a desired schema name.
cursor	Demonstrates the use of a cursor using static SQL. See “Example: Cursor Program” on page 82 for details.
delet	Demonstrates static SQL to delete items from a database.

Table 31. Embedded SQL Sample programs with No DB2 APIs (continued)

Sample Program Name	Program Description
dynamic	Demonstrates the use of a cursor using dynamic SQL. See “Example: Dynamic SQL Program” on page 136 for details.
fillcli	Demonstrates the client-side of a stored procedure that uses the SQLDA to pass information specifying which table the stored procedure populates with random data.
fillsrv	Demonstrates the server-side of a stored procedure example that uses the SQLDA to receive information from the client specifying the table that the stored procedure populates with random data.
inpccli	Demonstrates stored procedures using either the SQLDA structure or host variables. This is the client program of a client/server example. (The server program is called inpsrv.) The program fills the SQLDA with information, and passes it to the server program for further processing. The SQLCA status is returned to the client program. This program shows the invocation of stored procedures using an embedded SQL CALL statement. See “How the Example Input-SQLDA Client Application Works” on page 669 for details.
inpsrv	Demonstrates stored procedures using the SQLDA structure. This is the server program of a client/server example. (The client program is called inpccli.) The program creates a table (PRESIDENTS) in the sample database with the information received in the SQLDA. The server program does all the database processing and returns the SQLCA status to the client program. See “How the Example Input-SQLDA Stored Procedure Works” on page 675 for details.
joinsql	Demonstrates using advanced SQL join expressions.
largevol	Demonstrates parallel query processing in a partitioned environment, and the use of an NFS file system to automate the merging of the result sets. Only available on AIX. See “Example: Extracting Large Volume of Data (largevol.c)” on page 461 for details.
lobeval	Demonstrates the use of LOB locators and defers the evaluation of the actual LOB data. See “How the Sample LOBEVAL Program Works” on page 240 for details.
lobfile	Demonstrates the use of LOB file handles. See “How the Sample LOBFILE Program Works” on page 248 for details.
lobloc	Demonstrates the use of LOB locators. See “How the Sample LOBLOC Program Works” on page 231 for details.
lobval	Demonstrates the use of LOBs.
openftch	Demonstrates fetching, updating, and deleting of rows using static SQL. See “How the OPENFTCH Program Works” on page 91 for details.

Table 31. Embedded SQL Sample programs with No DB2 APIs (continued)

Sample Program Name	Program Description
outcli	Demonstrates stored procedures using a host variable. This is the client program of a client/server example. (The server program is called outsrv.) This program declares a single OUT variable to hold the value returned from the server program. This program demonstrates the use of the CREATE PROCEDURE statement to register a stored procedure and shows the invocation of stored procedures using an embedded SQL CALL statement. See “OUT Client Description” on page 202 for details.
outsrv	Demonstrates stored procedures using host variables. This is the server program of a client/server example. (The client program is called outcli.) The program sets the value of the function parameter to the median SALARY of the employees in the STAFF table of the sample database. The server does all the database processing (finding the median). The server program returns the host variable and the SQLCA status to the client program. See “OUT Stored Procedure Description” on page 206 for details.
recursql	Demonstrates the use of advanced SQL recursive queries.
sampudf	Demonstrates User-Defined Types (UDTs) and User-Defined Functions (UDFs) implemented to modify table entries. All UDFs declared in this program are sourced UDFs.
static	Demonstrates static SQL to retrieve information. See “Example: Static SQL Program” on page 61 for details.
tabsql	Demonstrates the use of advanced SQL table expressions.
tblcli	Demonstrates a call to a table function (client-side) to display weather information for a number of cities.
thdsrver	Demonstrates the use of POSIX threads APIs for thread creation and management. The program maintains a pool of contexts. A generate_work function is executed from main, and creates dynamic SQL statements that are executed by worker threads. When a context becomes available, a thread is created and dispatched to do the specified work. The work generated consists of statements to delete entries from either the STAFF or EMPLOYEE tables of the sample database. This program is only available on UNIX platforms.
trigsql	Demonstrates using advanced SQL triggers and constraints.
updat	Demonstrates static SQL to update a database. See “Example: UPDAT Program” on page 104 for details.
varinp	Demonstrates variable input to Embedded Dynamic SQL statement calls using parameter markers. See “How the VARINP Program Works” on page 167 for details.



---

## User-Defined Function Samples

Table 32. User-Defined Function Sample programs

Sample Program Name	Program Description
DB2Udf.java	A Java UDF that demonstrates several tasks, including integer division, manipulation of Character Large Objects (CLOBs), and the use of Java instance variables.
tblsrv.c	Demonstrates a table function (server-side) that processes weather information for a number of cities.
udf.c	Creates a library of User-Defined Functions (UDFs) made specifically for the sample database tables, but can be used with tables of compatible column types.
UDFsrv.java	Demonstrates the use of Java User-Defined Functions (UDFs).

---

## DB2 Call Level Interface Samples

Table 33. Sample CLI Programs in DB2 Universal Database

Sample Program Name	Program Description
<b>Utility files used by most CLI samples</b>	
samputil.c	Utility functions used by most samples
samputil.h	Header file for samputil.c, included by most samples
<b>General CLI Samples</b>	
adhoc.c	Interactive SQL with formatted output (was typical.c)
async.c	Run a function asynchronously (based on fetch.c)
basiccon.c	Basic connection
browser.c	List columns, foreign keys, index columns or stats for a table
calludf.c	Register and call a UDF
colpriv.c	List column Privileges
columns.c	List all columns for table search string
compnd.c	Compound SQL example
datasour.c	List all available data sources
descriptr.c	Example of descriptor usage
drivrcon.c	Rewrite of basiccon.c using SQLDriverConnect
duowcon.c	Multiple DUOW Connect type 2, syncpoint 1 (one phase commit)
embedded.c	Show equivalent DB2 CLI calls, for embedded SQL (in comments)
fetch.c	Simple example of a fetch sequence

Table 33. Sample CLI Programs in DB2 Universal Database (continued)

Sample Program Name	Program Description
getattrs.c	List some common environment, connection and statement options/attributes
getcurs.c	Show use of SQLGetCursor, and positioned update
getdata.c	Rewrite of fetch.c using SQLGetData instead of SQLBindCol
getfuncs.c	List all supported functions
getfuncs.h	Header file for getfuncs.c
getinfo.c	Use SQLGetInfo to get driver version and other information
getsqlca.c	Rewrite of adhoc.c to use prepare/execute and show cost estimate
lookres.c	Extract string from the resume clob field using locators
mixed.sqc	CLI sample with functions written using embedded SQL (Note: This file must be precompiled )
multicon.c	Multiple connections
native.c	Simple example of calling SQLNativeSql, and SQLNumParams
prepare.c	Rewrite of fetch.c, using prepare/execute instead of execdirect
proccols.c	List procedure parameters using SQLProcedureColumns
procs.c	List procedures using SQLProcedures
sfetch.c	Scrollable cursor example (based on xfetch.c)
setcolat.c	Set column attributes (using SQLSetColAttributes)
setcurs.c	Rewrite ofgetcurs.c using SQLSetCurs for positioned update
seteattrib.c	Set environment attribute (SQL_ATTR_OUTPUT_ANTS)
tables.c	List all tables
typeinfo.c	Display type information for all types for current data source
xfetch.c	Extended Fetch, multiple rows per fetch
<b>BLOB Samples</b>	
picin.c	Loads graphic BLOBS into the emp_photo table directly from a file using SQLBindParamToFile
picin2.c	Loads graphic BLOBS into the emp_photo table using SQLPutData
showpic.c	Extracts BLOB picture to file (using SQLBindColToFile), then displays the graphic.
showpic2.c	Extracts BLOB picture to file using piecewise output, then displays the graphic.
<b>Stored Procedure Samples</b>	
clicall.c	Defines a CLI function which is used in the embedded SQL sample mrspcli3.sqc
inpcli.c	Call embedded input stored procedure samples/c/inpsrv
inpcli2.c	Call CLI input stored procedure inpsrv2

Table 33. Sample CLI Programs in DB2 Universal Database (continued)

Sample Program Name	Program Description
inpsrv2.c	CLI input stored procedure (rewrite of embedded sample inpsrv.sqc)
mrspcli.c	CLI program that calls mrspsrv.c
mrspcli2.c	CLI program that calls mrspsrv2.sqc
mrspcli3.sqc	An embedded SQL program that calls mrspsrv2.sqc using clicall.c
mrspsrv.c	Stored procedure that returns a multi-row result set
mrspsrv2.sqc	An embedded SQL stored procedure that returns a multi-row result set
outcli.c	Call embedded output stored procedure samples/c/inpsrv
outcli2.c	Call CLI output stored procedure inpsrv2
outsrv2.c	CLI output stored procedure (rewrite of embedded sample inpsrv.sqc)
<b>Samples using ORDER tables created by create.c</b> (Run in the following order)	
create.c	Creates all tables for the order scenario
custin.c	Inserts customers into the customer table (array insert)
prodin.c	Inserts products into the products table (array insert)
prodpart.c	Inserts parts into the prod_parts table (array insert)
ordin.c	Inserts orders into the ord_line, ord_cust tables (array insert)
ordrep.c	Generates order report using multiple result sets
partrep.c	Generates exploding parts report (recursive SQL Query)
order.c	UDF library code (declares a 'price' UDF)
order.exp	Used to build order library
<b>Samples unchanged from DB2 Version 2</b>	
v2sutil.c	samputil.c using old v2 functions
v2sutil.h	samputil.h using old v2 functions
v2fetch.c	fetch.c using old v2 functions
v2xfetch.c	xfetch.c using old v2 functions

**Note:** Other files in the samples/cli directory include:

- README - Lists all example files.
- makefile - Makefile for all files
- build files for applications and stored procedures

## Java Samples

Table 34. Java Database Connectivity (JDBC) Sample Programs

Sample Program Name	Program Description
BasicCon.java	Basic connection.
Browser.java	List columns, foreign keys, index columns or stats for a table
ColPriv.java	List column Privileges
Columns.java	List all columns for table search string.
DB2Appl.java	A JDBC application that queries the sample database using the invoking user's privileges.
DB2Applt.java	A JDBC applet that queries the database using the JDBC applet driver. It uses the user name, password, server, and port number parameters specified in DB2Applt.html.
DB2Applt.html	An HTML file that embeds the applet sample program, DB2Applt. It needs to be customized with server and user information.
DB2SpCli.java	A Java client application that calls the JDBC stored procedure, DB2Stp.
DB2Stp.java	A Java stored procedure that updates the EMPLOYEE table on the server, and returns new salary and payroll information to the client.
DB2UdCli.java	A Java client application that calls the Java user-defined function, DB2Udf.
Dynamic.java	Demonstrates a cursor using dynamic SQL.
Embedded.java	Create, populate, list data from and drop a table.
GetAttrs.java	List some common environment, connection and statement options/attributes.
GetData.java	Simple querying and displaying of data from database.
Inpcli.java	A Java client application that calls the JDBC stored procedure, Inpsrv.
Inpsrv.java	A Java stored procedure demonstrating the SQLDA structure. It creates a PRESIDENTS table in the sample database with information received from the SQLDA, and returns the SQLCA status to the client application, Inpcli.
JavaSample.java	Creates the table, JAVA_SAMPLE, adds data, displays data, and deletes the table.
JobChange.java	Uses a statement and a prepared statement to query the database and update information at the same time (does the same as getcurs.c and setcurs.c but cannot do both ways therefore above method is used).
LookRes.java	Extract string from the resume clob field by converting the entire clob into a string, and searching for the desired substring.
MRSPcli.java	This is the client program that calls the server program MRSPsrv. The program demonstrates multiple result sets being returned from a Java stored procedure.
MRSPsrv.java	This is the server program that is called by the client program, MRSPcli. The program demonstrates multiple result sets being returned from a Java stored procedure.

Table 34. Java Database Connectivity (JDBC) Sample Programs (continued)

Sample Program Name	Program Description
MultiCon.java	Multiple connections.
Outcli.java	A Java client application that calls the SQLJ stored procedure, Outsrv.
PicIn.java	Loads graphic BLOBS into the emp_photo using the setBinaryStream method.
PluginEx.java	A Java program that adds new menu items and toolbar buttons to the DB2 Web Control Center.
Prepare.java	Rewrite of Simple.java, using a prepared statement.
ProcCols.java	List procedure parameters using getProcedureColumns under DatabaseMetaData.
Procs.java	List procedures using DatabaseMetaData.getProcedures.
ShowPic.java	Extracts BLOB picture to file using the getBinaryStream method. The file is read using the InputStream.read method.
Simple.java	Shows basic connecting querying and displaying the result set. (JDBC version of CLI sample fetch.c).
StpCli.java	A Java client application that calls the SQLJ stored procedure, Stp.
Tables.java	List all tables that match user defined search pattern.
Tools.java	Demonstrates a toolkit for Java programs written as DB2 samples.
TypeInfo.java	Display type information for all types for current data source.
UDFcli.java	A JDBC client application that calls functions in the Java user-defined function library, UDFsrv.
UsingThreads.java	Shows how to use threads to run an SQL statement asynchronously (JDBC version of CLI sample async.c).
Varinp.java	Demonstrates variable input to Embedded Dynamic SQL statement calls using parameter markers.
<b>Samples using ORDER tables created by Create.java (Run in the following order).</b>	
Create.java	Creates all tables for the order scenario.
CustIn.java	Inserts customers into the customer table (array insert).
ProdIn.java	Inserts products into the products table (array insert).
ProdPart.java	Inserts parts into the prod_parts table (array insert).
OrdIn.java	Inserts orders into the ord_line, ord_cust tables (array insert).
OrdRep.java	Generates a customer order report and uses the user defined function 'price' defined in order.c.
PartRep.java	Generates exploding parts report (recursive SQL Query).
DropJava.java	Removes the new tables created for the ORDER scenario.

Table 35. Embedded SQL for Java (SQLJ) Sample Programs

Sample Program Name	Program Description
App.sqlj	Uses static SQL to retrieve and update data from the EMPLOYEE table of the sample database.
Applt.sqlj	An applet that queries the database using the JDBC applet driver. It uses the user name, password, server, and port number parameters specified in Applt.html.
Applt.html	An HTML file that embeds the applet sample program, Applt. It needs to be customized with server and user information.
Cursor.sqlj	Demonstrates an iterator using static SQL.
OpF_Curs.sqlj	Class file for the Openftch program.
Openftch.sqlj	Demonstrates fetching, updating, and deleting rows using static SQL.
Outsrv.sqlj	Demonstrates a stored procedure using the SQLDA structure. It fills the SQLDA with the median salary of the employees in the STAFF table of the sample database. After the database processing (finding the median), the stored procedure returns the filled SQLDA and the SQLCA status to the JDBC client application, Outcli.
Static.sqlj	Uses static SQL to retrieve information.
Stp.sqlj	A stored procedure that updates the EMPLOYEE table on the server, and returns new salary and payroll information to the JDBC client program, StpCli.
UDFclie.sqlj	A client application that calls functions from the Java user-defined function library, UDFsrv.
Updat.sqlj	Uses static SQL to update a database.

## ADO, RDO, and MTS Samples

Table 36. ADO, RDO, and MTS Sample Programs

Sample Program Name	Program Description
Bank.vbp	An RDO program to create and maintain data for bank branches, with the ability to perform transactions on customer accounts. The program can use any database specified by the user as it contains the DDL to create the necessary tables for the application to store data.
Blob.vbp	This ADO program demonstrates retrieving BLOB data. It retrieves and displays pictures from the emp_photo table of the sample database. The program can also replace an image in the emp_photo table with one from a local file.
BLOBAccess.dsw	<p>This sample demonstrates highlighting ADO/Blob access using Microsoft Visual C++. It is similar to the Visual Basic sample, Blob.vbp. The BLOB sample has two main functions:</p> <ol style="list-style-type: none"> <li>1. Read a BLOB from the Sample database and display it to the screen.</li> <li>2. Read a BLOB from a file and insert it into the database. (Import)</li> </ol>

Table 36. ADO, RDO, and MTS Sample Programs (continued)

Sample Program Name	Program Description
Connect.vbp	This ADO program will create a connection object, and establish a connection, to the sample database. Once completed, the program will disconnect and exit.
Commit.vbp	This application demonstrates the use of autocommit/manual-commit features of ADO. The program queries the EMPLOYEE table of the sample database for employee number and name. The user has an option of connecting to the database in either autocommit or manual-commit mode. In the autocommit mode, all of the changes that a user makes on a record are updated automatically in the database. In the manual-commit mode, the user needs to begin a transaction before he/she can make any changes. The changes made since the beginning of a transaction can be undone by performing a rollback. The changes can be saved permanently by committing the transaction. Exiting the program automatically rolls back the changes.
db2com.vbp	<p>This Visual Basic project demonstrates updating a database using the Microsoft Transaction Server. It creates a server DLL used by the client program, db2mts.vbp, and has four class modules:</p> <ul style="list-style-type: none"> <li>• UpdateNumberColumn.cls</li> <li>• UpdateRow.cls</li> <li>• UpdateStringColumn.cls</li> <li>• VerifyUpdate.cls</li> </ul> <p>For this program a temporary table, DB2MTS, is created in the sample database.</p>
db2mts.vbp	This is a Visual Basic project for a client program that uses the Microsoft Transaction Server to call the server DLL created from db2com.vbp.
Select-Update.vbp	This ADO program performs the same functions as Connect.vbp, but also provides a GUI interface. With this interface, the user can view, update, and delete data stored in the ORG table of the sample database.
Sample.vbp	This Visual Basic project uses Keyset cursors via ADO to provide a graphical user interface to all data in the sample database.
VarCHAR.dsp	A Visual C++ program that uses ADO to access VarChar data as textfields. It provides a graphical user interface to allow users to view and update data in the ORG table of the sample database.

## Object Linking and Embedding Samples

Table 37. Object Linking and Embedding (OLE) Sample Programs

Sample Program Name	Program Description
sales.vbp	Demonstrates rollup queries on a Microsoft Excel sales spreadsheet (implemented in Visual Basic).
names.vbp	Queries a Lotus Notes address book (implemented in Visual Basic).

Table 37. Object Linking and Embedding (OLE) Sample Programs (continued)

Sample Program Name	Program Description
inbox.vbp	Queries Microsoft Exchange inbox e-mail messages through OLE/Messaging (implemented in Visual Basic).
invoice.vbp	An OLE automation user-defined function that sends Microsoft Word invoice documents as e-mail attachments (implemented in Visual Basic).
ccounter	A counter OLE automation user-defined function (implemented in Visual C++).
salarysrv.vbp	An OLE automation stored procedure that calculates the median salary of the STAFF table of the sample database (implemented in Visual Basic).
salaryclt	A client program that invokes the median salary OLE automation stored procedure salarysrv (implemented in Visual Basic and in Visual C++).

Table 38. Object Linking and Embedding Database (OLE DB) Table Functions

Sample Program Name	Program Description
jet.db2	Microsoft.Jet.OLEDB.3.51 Provider
mapi.db2	INTERSOLV Connect OLE DB for MAPI
msdaora.db2	Microsoft OLE DB Provider for Oracle
msdasql.db2	Microsoft OLE DB Provider for ODBC Drivers
msidxs.db2	Microsoft OLE DB Index Server Provider
notes.db2	INTERSOLV Connect OLE DB for Notes
sampprov.db2	Microsoft OLE DB Sample Provider
sqloledb.db2	Microsoft OLE DB Provider for SQL Server

## Command Line Processor Samples

Table 39. Command Line Processor (CLP) Sample Programs.

Sample File Name	File Description
const.db2	Creates a table with a CHECK CONSTRAINT clause.
cte.db2	Demonstrates a common table expression. The equivalent sample program demonstrating this advanced SQL statement is tabsql.
flt.db2	Demonstrates a recursive query. The equivalent sample program demonstrating this advanced SQL statement is recursql.
join.db2	Demonstrates an outer join of tables. The equivalent sample program demonstrating this advanced SQL statement is joinsql.
stock.db2	Demonstrates the use of triggers. The equivalent sample program demonstrating this advanced SQL statement is trigsq1.



Table 39. Command Line Processor (CLP) Sample Programs. (continued)

Sample File Name	File Description
testdata.db2	Uses DB2 built-in functions such as RAND() and TRANSLATE() to populate a table with randomly generated test data.
thaisort.db2	This script is particularly for Thai users. Thai sorting is by phonetic order requiring pre-sorting/swapping of the leading vowel and its consonant, as well as post-sorting in order to view the data in the correct sort order. The file implements Thai sorting by creating UDF functions presort and postsort, and creating a table; then it calls the functions against the table to sort the table data. To run this program, you first have to build the user-defined function program, udf, from the C source file, udf.c.

## Log Management User Exit Samples

Table 40. Log Management User Exit Sample Programs.

Sample File Name	File Description
db2uext2.cadsm	This is a sample User Exit utilizing ADSTAR DSM ( ADSM ) APIs to archive and retrieve database log files. The sample provides an audit trail of calls (stored in a separate file for each option) including a timestamp and parameters received. It also provides an error trail of calls in error including a timestamp and an error isolation string for problem determination. These options can be disabled. The file must be renamed db2uext2.c and compiled as a C program. Available on UNIX and Windows 32-bit operating systems. The OS/2 version is db2uexit.cad.
db2uexit.cad	This is the OS/2 version of db2uext2.cadsm. The file must be renamed db2uexit.c and compiled as a C program.
db2uext2.cdisk	This is a sample User Exit utilizing the system copy command for the particular platform on which it ships. The program archives and retrieves database log files, and provides an audit trail of calls (stored in a separate file for each option) including a timestamp and parameters received. It also provides an error trail of calls in error including a timestamp and an error isolation string for problem determination. These options can be disabled. The file must be renamed db2uext2.c and compiled as a C program. Available on UNIX and Windows 32-bit operating systems.
db2uext2.ctape	This is a sample User Exit utilizing system tape commands for the particular UNIX platform on which it ships. The program archives and retrieves database log files. All limitations of the system tape commands are limitations of this user exit. The sample provides an audit trail of calls (stored in a separate file for each option) including a timestamp and parameters received. It also provides an error trail of calls in error including a timestamp and an error isolation string for problem determination. These options can be disabled. The file must be renamed db2uext2.c and compiled as a C program. Available on UNIX platforms only.



---

## Appendix C. DB2DARI and DB2GENERAL Stored Procedures and UDFs

DB2DARI Stored Procedures . . . . .	655	COM.ibm.db2.app.StoredProc . . . . .	661
Using the SQLDA in a Client Application	655	COM.ibm.db2.app.UDF . . . . .	663
Using Host Variables in a DB2DARI Client . . . . .	656	COM.ibm.db2.app.Lob . . . . .	665
Using the SQLDA in a Stored Procedure	656	COM.ibm.db2.app.Blob . . . . .	665
Data Structure Manipulation . . . . .	657	COM.ibm.db2.app.Clob . . . . .	666
Summary of Data Structure Usage . . . . .	657	NOT FENCED Stored Procedures . . . . .	667
Input/Output SQLDA and SQLCA Structures . . . . .	658	Example Input-SQLDA Programs . . . . .	668
DB2GENERAL UDFs and Stored Procedures	659	How the Example Input-SQLDA Client Application Works . . . . .	669
Supported SQL Data Types . . . . .	659	FORTRAN UNIX Example: INPCLISQF . . . . .	671
Classes for Java Stored Procedures and UDFs . . . . .	661	How the Example Input-SQLDA Stored Procedure Works . . . . .	675
		REXX OS/2 Example: INPSRV.CMD . . . . .	676

This chapter describes how you can write DB2DARI and DB2GENERAL parameter style stored procedures and DB2GENERAL UDFs.

---

### DB2DARI Stored Procedures

When invoked, the DB2DARI stored procedure performs the following:

1. Accepts the SQLDA data structure from the client application. (Host variables are passed through an SQLDA data structure generated by the database manager when the SQL CALL statement is executed.)
2. Executes on the database server under the same transaction as the client application.
3. Returns SQLCA information and optional output data to the client application.

#### Using the SQLDA in a Client Application

To use the SQLDA structure to pass values to the stored procedure, perform the following steps before calling the stored procedure:

1. Allocate storage for the structure with the required number of base SQLVAR elements.
2. Set the SQLN field to the number of SQLVAR elements allocated.
3. Set the SQLD field to the number of SQLVAR elements actually used.
4. Initialize each SQLVAR element used as follows:
  - Set the SQLTYPE field to the proper data type.

- Set the SQLLEN field to the size of the data type.
- Allocate storage for the SQLDATA and SQLIND fields based upon the values in SQLTYPE and SQLLEN.

If your application will be working with character strings defined as FOR BIT DATA, you need to initialize the SQLDAID field to indicate that the SQLDA includes FOR BIT DATA definitions and the SQLNAME field of each SQLVAR that defines a FOR BIT DATA element.

If your application will be working with large objects, that is, data with types of CLOB, BLOB, or DBCLOB, you will also need to initialize the secondary SQLVAR elements. For information on the SQLDA structure, refer to the *SQL Reference*.

## Using Host Variables in a DB2DARI Client

Declare SQLVARs using the same approach discussed in “Allocating Host Variables” on page 189. In addition, the client application should set the indicator of output-only SQLVARs to -1 as discussed in “Data Structure Manipulation” on page 657. This will improve the performance of the parameter passing mechanism by avoiding having to pass the contents of the SQLDATA pointer, as only the indicator is sent. You should set the SQLTYPE field to a nullable data type for these parameters. If the SQLTYPE indicates a non-nullable data type, the indicator variable is not checked by the database manager.

## Using the SQLDA in a Stored Procedure

The stored procedure is invoked by the SQL CALL statement and executes using data passed to it by the client application. Information is returned to the client application using the stored procedure’s SQLDA structure.

The parameters of the SQL CALL statement are treated as both input and output parameters and are converted into the following format for the stored procedure:

```
SQL_API_RC SQL_API_FN proc_name(void *reserved1,
 void *reserved2,
 struct sqlda *inoutsqlda,
 struct sqlca *sqlca)
```

The SQL\_API\_FN is a macro that specifies the calling convention for a function that may vary across each supported operating system. This macro is required when you write stored procedures or UDFs.

Following is an example of how a CALL statement maps to a server’s parameter list:

```
CALL OUTSRV (:empno:empind,:salary:salind)
```

The parameters to this call are converted into an SQLDA structure with two SQLVARs. The first SQLVAR points to the empno host variable and the empind indicator variable. The second SQLVAR points to the salary host variable and the salind indicator variable.

**Note:** The SQLDA structure is not passed to the stored procedure if the number of elements, SQLD, is set to 0. In this case, if the SQLDA is not passed, the stored procedure receives a NULL pointer.

### Data Structure Manipulation

The database manager automatically allocates a duplicate SQLDA structure at the database server. To reduce network traffic, it is important to indicate which host variables are input-only, and which ones are output-only. The client procedure should set the indicator of output-only SQLVARs to -1. The server procedure should set the indicator for input-only SQLVARs to -128. This allows the database manager to choose which SQLVARs are passed.

Note that an indicator variable is not reset if the client or the server sets it to a negative value (indicating that the SQLVAR should not be passed). If the host variable to which the SQLVAR refers is given a value in the stored procedure or the client code, its indicator variable should be set to zero or a positive value so that the value is passed. For example, consider a stored procedure which takes one output-only parameter, called as follows:

```
empind = -1;
EXEC SQL CALL storproc(:empno:empind);
```

When the stored procedure sets the value for the first SQLVAR, it should also set the value of the indicator to a non-negative value so that the result is passed back to empno.

### Summary of Data Structure Usage

Table 41 summarizes the use of the various structure fields by the stored procedures application. In the table, sqlda is an SQLDA structure passed to the stored procedure and n is a numeric value indicating a specific SQLVAR element of the SQLDA. The numbers on the right refer to the notes following the table.

Table 41. Stored Procedures Parameter Variables

Input/Output SQLDA	sqlda.SQLDAID		4
	sqlda.SQLDABC		4
	sqlda.SQLEN	2	4
	sqlda.SQLD	2 3	5

Table 41. Stored Procedures Parameter Variables (continued)

Input/Output SQLVAR	sqlda.n.SQLTYPE	2	3	5		
	sqlda.n.SQLEN	2	3	5		
	sqlda.n.SQLDATA	1	2	3	6	8
	sqlda.n.SQLIND	1	2	3	6	8 9
	sqlda.n.SQLNAME.length				6	7
	sqlda.n.SQLNAME.data				6	7
	sqlda.n.SQLDATATYPE_NAME	2	3	5		
	sqlda.n.SQLLONGLEN	2	3	5		
	sqlda.n.SQLDATALEN	1	2	3	6	7
SQLCA (all elements)					6	7

**Note:**

Before invoking the stored procedure, the client application must:

1. Allocate storage for the pointer element based on SQLTYPE and SQLEN.
2. Initialize the element with the appropriate data.

When called by the application, the database manager:

3. Sends data in the original element to a duplicate element allocated at the stored procedure. The SQLN element is initialized with the data in the SQLD element.

When invoked, the stored procedure can:

4. Alter data in the duplicate element. The data can be altered as needed since it is not checked for validity or returned to the client application.

When the stored procedure terminates, the database manager:

5. Checks data in the duplicate elements. If the values in these fields do not match the data in the original elements, an error is returned.
6. Returns data in the duplicate elements to the original element.
7. The data can be altered as needed since it is not checked for validity.
8. The data pointed to by the elements can be altered as needed since they are not checked for validity but are returned to the client application.
9. The SQLIND field is not passed in or out if SQLTYPE indicates the column type is not nullable.

## Input/Output SQLDA and SQLCA Structures

The stored procedure runs using any information passed in the input variables of the SQLDA structure. Information is returned to the client in the output variables of the SQLDA. Do not change the value of the SQLD, SQLTYPE, and SQLEN fields of the SQLDA, as these fields are compared to the original values set by the client application before data is returned. If they are different, one of the following SQLCODEs is returned:

**SQLCODE -1113 (SQLSTATE 39502)**

The data type of a variable (that is, the value in SQLTYPE) has changed.

**SQLCODE -1114 (SQLSTATE 39502)**

The length of a variable (that is, the value in SQLLEN) has changed.

**SQLCODE -1115 (SQLSTATE 39502)**

The SQLD field has changed.

In addition, do not change the pointer for the SQLDATA and the SQLIND fields, although you can change the value that is pointed to by these fields.

**Note:** It is possible to use the same variable for both input and output.

Before the stored procedure returns, SQLCA information should be explicitly copied to the SQLCA parameter of the stored procedure.

---

**DB2GENERAL UDFs and Stored Procedures**

PARAMETER STYLE DB2GENERAL UDFs and stored procedures are written in Java, and hereafter are referred to simply as Java UDFs and stored procedures. Creating DB2GENERAL UDFs and stored procedures is very similar to creating UDFs and stored procedures in other supported programming languages. Once you have created and registered them, you can call them from programs in any language. Typically, you may call JDBC APIs from your stored procedures, but you can not call them from UDFs.

**Supported SQL Data Types**

When you call PARAMETER STYLE DB2GENERAL UDFs and stored procedures, DB2 converts SQL types to and from Java types for you as described in Table 42. Several of these classes are provided in the Java package `COM.ibm.db2.app`.

Table 42. DB2 SQL Types and Java Objects

SQL Column Type	Java Type (UDF)	Java Type (Stored Procedure)
SMALLINT (500/501)	short	short
INTEGER (496/497)	int	int
BIGINT (492/493)	long	long
FLOAT (480/481)	double	double
REAL (480/481) <sup>1</sup>	float	float
DECIMAL(p,s) (484/485)	java.math.BigDecimal	java.math.BigDecimal

Table 42. DB2 SQL Types and Java Objects (continued)

SQL Column Type	Java Type (UDF)	Java Type (Stored Procedure)
NUMERIC(p,s) (504/505)	java.math.BigDecimal	java.math.BigDecimal
CHAR(n) (452/453)	String	String
CHAR(n) FOR BIT DATA (452/453)	Blob	Blob
C null-terminated string (400/401) <sup>2</sup>	n/a	String
VARCHAR(n)(448/449)	String	String
VARCHAR(n) FOR BIT DATA (448/449)	Blob	Blob
LONG VARCHAR (456/457)	String	String
LONG VARCHAR FOR BIT DATA (456/457)	Blob	Blob
GRAPHIC(n) (468/469)	String	String
C null-terminated graphic string (460/461) <sup>2</sup>	n/a	String
VARGRAPHIC(n) (464/465)	String	String
LONG VARGRAPHIC (472/473) <sup>3</sup>	String	String
BLOB(n)(404/405) <sup>3</sup>	Blob	Blob
CLOB(n) (408/409) <sup>3</sup>	Clob	Clob
DBCLOB(n) (412/413) <sup>3</sup>	Clob	Clob
DATE (384/385) <sup>4</sup>	String	String
TIME (388/389) <sup>4</sup>	String	String
TIMESTAMP (392/393) <sup>4</sup>	String	String

**Notes:**

1. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
2. Parenthesized types, such as the C null-terminated graphic string, occur in stored procedures when the calling application uses embedded SQL with some host variable types.
3. The Blob and Clob classes are provided in the `COM.ibm.db2.app` package. Their interfaces include routines to generate an `InputStream` and `OutputStream` for reading from and writing to a Blob, and a `Reader` and `Writer` for a Clob. See “Classes for Java Stored Procedures and UDFs” on page 661 for descriptions of the classes.
4. SQL DATE, TIME, and TIMESTAMP values use the ISO string encoding in Java, as they do for UDFs coded in C.

Instances of classes `COM.ibm.db2.app.Blob` and `COM.ibm.db2.app.Clob` represent the LOB data types (BLOB, CLOB, and DBCLOB). These classes provide a limited interface to read LOBs passed as inputs, and write LOBs returned as outputs. Reading and writing of LOBs occur through standard Java I/O stream objects. For the Blob class, the routines `getInputStream()` and



`getOutputStream()` return an `InputStream` or `OutputStream` object through which the BLOB content may be processed bytes-at-a-time. For a Clob, the routines `getReader()` and `getWriter()` will return a `Reader` or `Writer` object through which the CLOB or DBCLOB content may be processed characters-at-a-time.

If such an object is returned as an output using the `set()` method, code page conversions may be applied in order to represent the Java Unicode characters in the database code page.

## Classes for Java Stored Procedures and UDFs

Java stored procedures are very similar to Java UDFs. Like table functions, they can have multiple outputs. They also use the same conventions for NULL values, and the same set routine for output. The main difference is that a Java class that contains stored procedures must inherit from the `COM.ibm.db2.app.StoredProc` class instead of the `COM.ibm.db2.app.UDF` class. Refer to “`COM.ibm.db2.app.StoredProc`” for a description of the `COM.ibm.db2.app.StoredProc` class.

This interface provides the following routine to fetch a JDBC connection to the embedding application context:

```
public java.sql.Connection getConnection()
```

You can use this handle to run SQL statements. Other methods of the `StoredProc` interface are listed in the file `sqllib/samples/java/StoredProc.java`.

There are five classes/interfaces that you can use with Java Stored Procedures or UDFs:

- `COM.ibm.db2.app.StoredProc`
- `COM.ibm.db2.app.UDF`
- `COM.ibm.db2.app.Lob`
- `COM.ibm.db2.app.Blob`
- `COM.ibm.db2.app.Clob`

The following sections describe the public aspects of these classes' behavior:

### **COM.ibm.db2.app.StoredProc**

A Java class that contains methods intended to be called as PARAMETER STYLE DB2GENERAL stored procedures must be public and must implement this Java interface. You must declare such a class as follows:

```
public class <user-STP-class> extends COM.ibm.db2.app.StoredProc{ ... }
```

You can only call inherited methods of the `COM.ibm.db2.app.StoredProc` interface in the context of the currently executing stored procedure. For example, you cannot use operations on LOB arguments, result- or status-setting calls, etc., after a stored procedure returns. A Java exception will be thrown if you violate this rule.

Argument-related calls use a column index to identify the column being referenced. These start at 1 for the first argument. All arguments of a `PARAMETER STYLE DB2GENERAL` stored procedure are considered INOUT and thus are both inputs and outputs.

Any exception returned from the stored procedure is caught by the database and returned to the caller with `SQLCODE -4302`, `SQLSTATE 38501`. A JDBC `SQLException` or `SQLWarning` is handled specially and passes its own `SQLCODE`, `SQLSTATE` etc. to the calling application verbatim.

The following methods are associated with the `COM.ibm.db2.app.StoredProc` class:

```
public StoredProc() [default constructor]
```

This constructor is called by the database before the stored procedure call.

```
public boolean isNull(int) throws Exception
```

This function tests whether an input argument with the given index is an SQL NULL.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

This function sets the output argument with the given index to the given value. The index has to refer to a valid output argument, the data type must match, and the value must have an acceptable length and contents. Strings with Unicode characters must be representable in the database code page. Errors result in an exception being thrown.

```
public java.sql.Connection getConnection() throws Exception
```

This function returns a JDBC object that represents the calling application's connection to the database. It is analogous to the result of a `null SQLConnect()` call in a C stored procedure.

## COM.ibm.db2.app.UDF

A Java class that contains methods intended to be called as PARAMETER STYLE DB2GENERAL UDFs must be public and must implement this Java interface. You must declare such a class as follows:

```
public class <user-UDF-class> extends COM.ibm.db2.app.UDF{ ... }
```

You can only call methods of the COM.ibm.db2.app.UDF interface in the context of the currently executing UDF. For example, you cannot use operations on LOB arguments, result- or status-setting calls, etc., after a UDF returns. A Java exception will be thrown if this rule is violated.

Argument-related calls use a column index to identify the column being set. These start at 1 for the first argument. Output arguments are numbered higher than the input arguments. For example, a scalar UDF with three inputs uses index 4 for the output.

Any exception returned from the UDF is caught by the database and returned to the caller with SQLCODE -4302, SQLSTATE 38501.

The following methods are associated with the COM.ibm.db2.app.UDF class:

```
public UDF() [default constructor]
```

This constructor is called by the database at the beginning of a series of UDF calls. It precedes the first call to the UDF.

```
public void close()
```

This function is called by the database at the end of a UDF evaluation, if the UDF was created with the FINAL CALL option. It is analogous to the final call for a C UDF. For table functions, close() is called after the CLOSE call to the UDF method (if NO FINAL CALL is coded or defaulted), or after the FINAL call (if FINAL CALL is coded). If a Java UDF class does not implement this function, a no-op stub will handle and ignore this event.

```
public int getCallType() throws Exception
```

Table function UDF methods use getCallType() to find out the call type for a particular call. It returns a value as follows (symbolic defines are provided for these values in the Com.ibm.db2.app.UDF class definition):

- -2 FIRST call
- -1 OPEN call
- 0 FETCH call
- 1 CLOSE call
- 2 FINAL call

```
public boolean isNull(int) throws Exception
```

This function tests whether an input argument with the given index is an SQL NULL.

```
public boolean needToSet(int) throws Exception
```

This function tests whether an output argument with the given index needs to be set. This may be false for a table UDF declared with DBINFO, if that column is not used by the UDF caller.

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

This function sets the output argument with the given index to the given value. The index has to refer to a valid output argument, the data type must match, and the value must have an acceptable length and contents. Strings with Unicode characters must be representable in the database code page. Errors result in an exception being thrown.

```
public void setSQLstate(String) throws Exception
```

This function may be called from a UDF to set the SQLSTATE to be returned from this call. A table UDF should call this function with "02000" to signal the end-of-table condition. If the string is not acceptable as an SQLSTATE, an exception will be thrown.

```
public void setSQLmessage(String) throws Exception
```

This function is similar to the setSQLstate function. It sets the SQL message result. If the string is not acceptable (for example, longer than 70 characters), an exception will be thrown.

```
public String getFunctionName() throws Exception
```

This function returns the name of the executing UDF.

```
public String getSpecificName() throws Exception
```

This function returns the specific name of the executing UDF.

```
public byte[] getDBinfo() throws Exception
```

This function returns a raw, unprocessed DBINFO structure for the executing UDF, as a byte array. You must first declare it with the DBINFO option.

```
public String getDBname() throws Exception
public String getDBauthid() throws Exception
public String getDBtbschema() throws Exception
public String getDBtbname() throws Exception
```

```

public String getDBcolname() throws Exception
public String getDBver_rel() throws Exception
public String getDBplatform() throws Exception
public String getDBapplid() throws Exception

```

These functions return the value of the appropriate field from the DBINFO structure of the executing UDF.

```

public int[] getDBcodepg() throws Exception

```

This function returns the SBCS, DBCS, and composite code page numbers for the database, from the DBINFO structure. The returned integer array has the respective numbers as its first three elements.

```

public byte[] getScratchpad() throws Exception

```

This function returns a copy of the scratchpad of the currently executing UDF. You must first declare the UDF with the SCRATCHPAD option.

```

public void setScratchpad(byte[]) throws Exception

```

This function overwrites the scratchpad of the currently executing UDF with the contents of the given byte array. You must first declare the UDF with the SCRATCHPAD option. The byte array must have the same size as getScratchpad() returns.

### **COM.ibm.db2.app.Lob**

This class provides utility routines that create temporary Blob or Clob objects for computation inside user-defined functions or stored procedures.

The following methods are associated with the COM.ibm.db2.app.Lob class:

```

public static Blob newBlob() throws Exception

```

This function creates a temporary Blob. It will be implemented using a LOCATOR if possible.

```

public static Clob newClob() throws Exception

```

This function creates a temporary Clob. It will be implemented using a LOCATOR if possible.

### **COM.ibm.db2.app.Blob**

An instance of this class is passed by the database to represent a BLOB as UDF or stored procedure input, and may be passed back as output. The application may create instances, but only in the context of an executing UDF or stored procedure. Uses of these objects outside such a context will throw an exception.

The following methods are associated with the `COM.ibm.db2.app.Blob` class:

`public long size() throws Exception`

This function returns the length (in bytes) of the BLOB.

`public java.io.InputStream getInputStream() throws Exception`

This function returns a new `InputStream` to read the contents of the BLOB. Efficient seek/mark operations are available on that object.

`public java.io.OutputStream getOutputStream() throws Exception`

This function returns a new `OutputStream` to append bytes to the BLOB. Appended bytes become immediately visible on all existing `InputStream` instances produced by this object's `getInputStream()` call.

### **COM.ibm.db2.app.Clob**

An instance of this class is passed by the database to represent a CLOB or DBCLOB as UDF or stored procedure input, and may be passed back as output. The application may create instances, but only in the context of an executing UDF or stored procedure. Uses of these objects outside such a context will throw an exception.

Clob instances store characters in the database code page. Some Unicode characters may not be representable in this code page, and may cause an exception to be thrown during conversion. This may happen during an append operation, or during a UDF or `StoredProc.set()` call. This is necessary to hide the distinction between a CLOB and a DBCLOB from the Java programmer.

The following methods are associated with the `COM.ibm.db2.app.Clob` class:

`public long size() throws Exception`

This function returns the length (in characters) of the CLOB.

`public java.io.Reader getReader() throws Exception`

This function returns a new `Reader` to read the contents of the CLOB or DBCLOB. Efficient seek/mark operations are available on that object.

`public java.io.Writer getWriter() throws Exception`

This function returns a new `Writer` to append characters to this CLOB or DBCLOB. Appended characters become immediately visible on all existing `Reader` instances produced by this object's `GetReader()` call.

## NOT FENCED Stored Procedures

To indicate that a DB2DARI stored procedure should run as a NOT FENCED stored procedure, place it in the directory indicated in the *Application Building Guide*. For more information on NOT FENCED stored procedures, see “NOT FENCED Stored Procedures” on page 210.

## Example Input-SQLDA Programs

Following is a sample program demonstrating the use of an input SQLDA structure. The client application invokes a stored procedure that creates a table named Presidents and loads the table with data.

This program creates a table called Presidents in the SAMPLE database. It then inserts the values Washington, Jefferson, and Lincoln into the table.

Without using stored procedures, the sample program would have been designed to transmit data across the network in four separate requests in order to process each SQL statement, as shown in Figure 11.

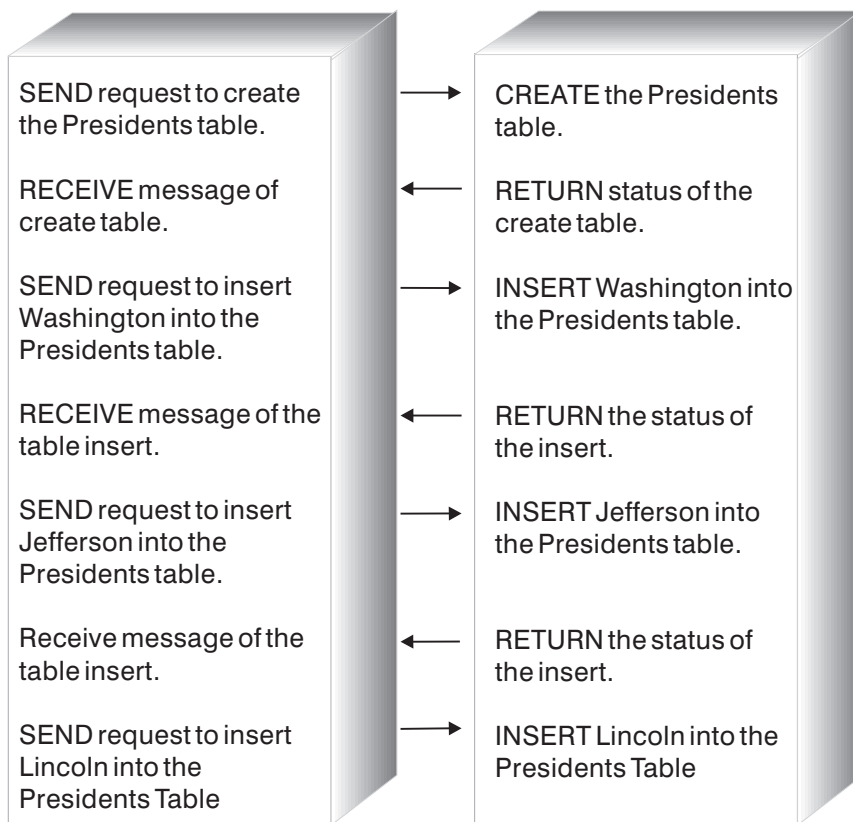


Figure 11. Input-SQLDA Sample Without a Stored Procedure

Instead, the sample program makes use of the stored procedures technique to transmit all of the data across the network in one request, allowing the server



procedure to execute the SQL statements as a group. This technique is shown in Figure 12.

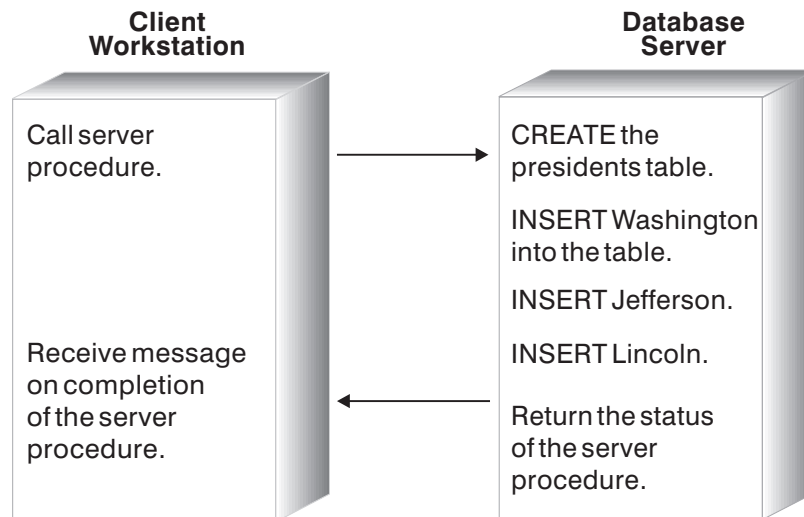


Figure 12. Input-SQLDA Sample With a Stored Procedure

A sample input-SQLDA client application and sample input-SQLDA stored procedure is shown on 669.

### How the Example Input-SQLDA Client Application Works

1. **Initialize the Input SQLDA Structure.** The following fields of the input SQLDA are initialized:
  - The SQLN and SQLD elements are set to the total number of SQLVAR elements allocated and used.
  - The SQLTYPE elements are set to indicate character data type.
  - The first SQLDATA element is set to the name of the table. The second through fourth SQLDATA elements are set to the values Washington, Jefferson, and Lincoln.
  - The SQLLEN elements are set to the length of each SQLDATA element (plus 1 byte for the C language null terminator).
  - The SQLIND elements are set to NULL.
2. **Invoke the Server Procedure.** The application invokes the procedure `inpsrv` at the location of the database, sample using:
  - a. `CALL` statement with host variables
  - b. `CALL` statement with an SQLDA.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

- C**                    `check_error` is redefined as CHECKERR and is located in the `util.c` file.
- COBOL**            CHECKERR is an external program named `checkerr.cbl`.
- FORTTRAN**        CHECKERR is a subroutine located in the `util.f` file.
- REXX**             CHECKERR is a procedure located at bottom of the current program.

See “Using GET ERROR MESSAGE in Example Programs” on page 120 for the source code for this error checking utility.

## FORTRAN UNIX Example: INPCLI.SQF

```
program inpccli
implicit none

* Copy Files for Constants and Structures
include 'sql.f'
include 'sqlenv.f'
include 'sqlutil.f'
include 'sqldact.f'

* Declare an SQLCA
EXEC SQL INCLUDE SQLCA

* Declare host variables.
EXEC SQL BEGIN DECLARE SECTION
 character*8 dbname
 character*8 userid
 character*18 passwd
 character*11 procname /'inpsrv'/
 character*10 table_name /'PRESIDENTS'/
 character*20 data_item0 /'Washington'/
 character*20 data_item1 /'Jefferson'/
 character*20 data_item2 /'Lincoln'/
 integer*2 tableind /0/
 integer*2 dataind0 /0/
 integer*2 dataind1 /0/
 integer*2 dataind2 /0/
EXEC SQL END DECLARE SECTION

* Declare Variables Used to Create an SQLDA
integer*2 sqlvar1
parameter (sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz)
integer*2 sqlvar2
parameter (sqlvar2 = sqlda_header_sz + 1*sqlvar_struct_sz)
integer*2 sqlvar3
parameter (sqlvar3 = sqlda_header_sz + 2*sqlvar_struct_sz)
integer*2 sqlvar4
parameter (sqlvar4 = sqlda_header_sz + 3*sqlvar_struct_sz)

* Declare an Input SQLDA Structure -- 4 Variables
character io_sqlda(sqlda_header_sz + 4*sqlvar_struct_sz)

character*8 io_sqldaaid ! Header
integer*4 io_sqldabc
integer*2 io_sqln
integer*2 io_sqld

integer*2 io_sqltype1 ! First Variable
integer*2 io_sqlllen1
integer*4 io_sqldata1
integer*4 io_sqlind1
integer*2 io_sqlnamel1
character*30 io_sqlnamec1

integer*2 io_sqltype2 ! Second Variable
```

```

integer*2 io_sqllen2
integer*4 io_sqldata2
integer*4 io_sqlind2
integer*2 io_sqlname12
character*30 io_sqlnamec2

integer*2 io_sqltype3 ! Third Variable
integer*2 io_sqllen3
integer*4 io_sqldata3
integer*4 io_sqlind3
integer*2 io_sqlname13
character*30 io_sqlnamec3

integer*2 io_sqltype4 ! Fourth Variable
integer*2 io_sqllen4
integer*4 io_sqldata4
integer*4 io_sqlind4
integer*2 io_sqlname14
character*30 io_sqlnamec4

equivalence(io_sqlda(sqlda_sqldaids_ofs), io_sqldaids)
equivalence(io_sqlda(sqlda_sqldabc_ofs), io_sqldabc)
equivalence(io_sqlda(sqlda_sqln_ofs), io_sqln)
equivalence(io_sqlda(sqlda_sqld_ofs), io_sqld)

equivalence(io_sqlda(sqlvar1+sqlvar_type_ofs), io_sqltype1)
equivalence(io_sqlda(sqlvar1+sqlvar_len_ofs), io_sqllen1)
equivalence(io_sqlda(sqlvar1+sqlvar_data_ofs), io_sqldata1)
equivalence(io_sqlda(sqlvar1+sqlvar_ind_ofs), io_sqlind1)
equivalence(io_sqlda(sqlvar1+sqlvar_name_length_ofs),
+ io_sqlname11)
equivalence(io_sqlda(sqlvar1+sqlvar_name_data_ofs),
+ io_sqlnamec1)

equivalence(io_sqlda(sqlvar2+sqlvar_type_ofs), io_sqltype2)
equivalence(io_sqlda(sqlvar2+sqlvar_len_ofs), io_sqllen2)
equivalence(io_sqlda(sqlvar2+sqlvar_data_ofs), io_sqldata2)
equivalence(io_sqlda(sqlvar2+sqlvar_ind_ofs), io_sqlind2)
equivalence(io_sqlda(sqlvar2+sqlvar_name_length_ofs),
+ io_sqlname12)
equivalence(io_sqlda(sqlvar2+sqlvar_name_data_ofs),
+ io_sqlnamec2)

equivalence(io_sqlda(sqlvar3+sqlvar_type_ofs), io_sqltype3)
equivalence(io_sqlda(sqlvar3+sqlvar_len_ofs), io_sqllen3)
equivalence(io_sqlda(sqlvar3+sqlvar_data_ofs), io_sqldata3)
equivalence(io_sqlda(sqlvar3+sqlvar_ind_ofs), io_sqlind3)
equivalence(io_sqlda(sqlvar3+sqlvar_name_length_ofs),
+ io_sqlname13)
equivalence(io_sqlda(sqlvar3+sqlvar_name_data_ofs),
+ io_sqlnamec3)

equivalence(io_sqlda(sqlvar4+sqlvar_type_ofs), io_sqltype4)
equivalence(io_sqlda(sqlvar4+sqlvar_len_ofs), io_sqllen4)
equivalence(io_sqlda(sqlvar4+sqlvar_data_ofs), io_sqldata4)

```

```

 equivalence(io_sqlda(sqlvar4+sqlvar_ind_ofs), io_sqlind4)
 equivalence(io_sqlda(sqlvar4+sqlvar_name_length_ofs),
+ io_sqlname14)
 equivalence(io_sqlda(sqlvar4+sqlvar_name_data_ofs),
+ io_sqlnamec4)

 character*80 errloc
 integer*4 rc

* Program Logic
 print *, 'Enter in the database name : '
 read 100, dbname
100 format (a8)

 print *, 'Enter your user id (default none):'
 read 100, userid

 if(userid(1:1) .eq. ' ') then
EXEC SQL CONNECT TO :dbname
 else
 print *, 'Enter your password : '
 read 100, passwd

 print *, 'CONNECT to Remote Database.'

EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
 end if

* Connect to Remote Database
 errloc = 'CONNECT'
 call checkerr (sqlca, errloc, *999)

* Call the Remote Procedure via CALL with Host Variables
* 2a
 print *, 'Use CALL with Host Variable '
 print *, 'to invoke the Server Procedure named inpsrv.'
 EXEC SQL CALL :procname (:table_name:tableind,
+ :data_item0:dataind0,
+ :data_item1:dataind1,
+ :data_item2:dataind2)
 errloc = 'CALL with HOST VARIABLES'
 call checkerr (sqlca, errloc, *999)
 print *, 'Server Procedure Complete.'

* Initialize the Input SQLDA Structure
* 1
 io_sqlda_id = 'IN_SQLDA'
 io_sqlda_bc = sqlda_header_sz + 4*sqlvar_struct_sz
 io_sqln = 4
 io_sqld = 4

 io_sqltype1 = SQL_TYP_NCHAR
 io_sqlllen1 = 10
 io_sqlind1 = 0
 rc = sqlgaddr(%ref(table_name), %ref(io_sqldata1))

```

```

rc = sqlgaddr(%ref(tableind), %ref(io_sqlind1))

io_sqltype2 = SQL_TYP_NCHAR
io_sqlllen2 = 20
io_sqlind2 = 0
rc = sqlgaddr(%ref(data_item0), %ref(io_sqldata2))
rc = sqlgaddr(%ref(dataind0), %ref(io_sqlind2))

io_sqltype3 = SQL_TYP_NCHAR
io_sqlllen3 = 20
io_sqlind3 = 0
rc = sqlgaddr(%ref(data_item1), %ref(io_sqldata3))
rc = sqlgaddr(%ref(dataind1), %ref(io_sqlind3))

io_sqltype4 = SQL_TYP_NCHAR
io_sqlllen4 = 20
io_sqlind4 = 0
rc = sqlgaddr(%ref(data_item2), %ref(io_sqldata4))
rc = sqlgaddr(%ref(dataind2), %ref(io_sqlind4))

* Call the Remote Procedure via CALL with SQLDA
* 2b
print *, 'Use CALL with SQLDA to invoke the Server Procedure named
cinpsrv.'
EXEC SQL CALL :procname USING DESCRIPTOR :io_sqlda
errloc = 'CALL with SQLDA'
call checkerr (sqlca, errloc, *999)
print *, 'Server Procedure Complete.'

* Disconnect from Remote Database.
EXEC SQL CONNECT RESET
errloc = 'CONNECT RESET'
call checkerr (sqlca, errloc, *999)
999 stop
end

```

## How the Example Input-SQLDA Stored Procedure Works

1. **Declare Server Procedure.** The procedure accepts pointers to SQLDA and SQLCA structures.
2. **Create Table.** Using the data passed in the first SQLVAR of the SQLDA structure, a CREATE TABLE statement is constructed and executed to create a table named Presidents.
3. **Prepare Insert Statement.** An INSERT statement with a parameter marker ? is prepared.
4. **Insert Data.** The INSERT statement prepared previously is executed using the data passed in the second through fourth SQLVAR of the SQLDA structure. The parameter markers are replaced with the values Washington, Jefferson, and Lincoln. These values are inserted into the Presidents table.
5. **Return to the Client Application.** The server procedure copies the SQLCA to the SQLCA of the client application, issues a COMMIT statement if the transaction is successful, and returns the value SQLZ\_DISCONNECT\_PROC, indicating that no further calls to the server procedure will be made.

**Note:** Server procedures cannot be written in REXX on AIX systems.

## REXX OS/2 Example: INPSRV.CMD

```
/* REXX INPut SeRVer */

/* this variable (SYSTEM) must be user defined */
SYSTEM = NT
if SYSTEM = NT then do
 if rxfuncquery('SQLDBS') <> 0 then
 rcy = rxfuncadd('SQLDBS', 'DB2AR', 'SQLDBS')

 if rxfuncquery('SQLEXEC') <> 0 then
 rcy = rxfuncadd('SQLEXEC', 'DB2AR', 'SQLEXEC')
end

if SYSTEM = AIX then
 rcy = SysAddFuncPkg("/usr/lpp/db2_05_00/lib/db2rex")

table_stmt = 'CREATE TABLE ' || sqlrida.1.sqldata || ' (name CHAR(20))' 2
call SQLEXEC 'EXECUTE IMMEDIATE :table_stmt'
if sqlca.sqlcode = -601 then
 sqlca.sqlcode = 0
call CHECKERR 'EXECUTE IMMEDIATE CREATE TABLE'

insert_stmt = 'INSERT INTO ' || sqlrida.1.sqldata || ' VALUES (?)'
call SQLEXEC 'PREPARE s1 FROM :insert_stmt' 3
call CHECKERR 'PREPARE INSERT STATEMENT'

cntr = 1
do while (cntr < SQLRIDA.sqld & sqlca.sqlcode = 0)
 cntr = cntr + 1
 insert_data = SQLRIDA.cntr.sqldata
 call SQLEXEC 'EXECUTE s1 USING :insert_data' 4
end
call CHECKERR 'EXECUTION OF INSERT STATEMENT'

call SQLEXEC 'COMMIT'
call CHECKERR 'COMMIT'

exit 0 5

CHECKERR:
 arg errloc

 if (SQLCA.SQLCODE = 0) then
 return 0
 else
 say '--- error report ---'
 say 'ERROR ocured :' errloc
 say 'SQLCODE :' SQLCA.SQLCODE

 /*****\
 * GET ERROR MESSAGE API called *
 *****/
```



```

call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
say errmsg
say '--- end error report ---'

if (SQLCA.SQLCODE < 0) then
 exit
else
 say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
 return 0
end
end
return 0

```



---

## Appendix D. Programming in a Host or AS/400 Environment

This section contains information that is common to the *DB2 Connect User's Guide*. If you encounter an unfamiliar term or concept in this section, refer to the *DB2 Connect User's Guide*.

DB2 Connect lets an application program access data in any host or AS/400 server database. For example, an application running on OS/2 can access data in a DB2 Universal Database for OS/390 database. You can create new applications, or modify existing applications to run in a host or AS/400 environment. You can also develop applications in one environment and port them to another.

DB2 Connect enables you to use the following items with host database products such as DB2 Universal Database for OS/390, as long as the item is supported by the host database product:

- Embedded SQL, both static and dynamic
- Compound SQL, as described in “NOT ATOMIC Compound SQL” on page 690
- Stored procedures, as described in “Stored Procedures” on page 689
- The DB2 Call Level Interface
- The Microsoft ODBC API
- JDBC.

Some SQL statements differ among relational database products. You may encounter the following situations:

- SQL statements that are the same for all the database products that you use regardless of standards
- SQL statements that are documented in the *SQL Reference* and are therefore available in all IBM relational database products
- SQL statements that are unique to one database system that you access.

SQL statements in the first two categories are highly portable, but those in the third category will first require changes.

In general, SQL statements in Data Definition Language (DDL) are not as portable as those in Data Manipulation Language (DML).

DB2 Connect accepts some SQL statements that are not supported by DB2 Universal Database. DB2 Connect passes these statements on to the host or AS/400 server.

For information on limits on different platforms, such as the maximum column length, refer to the *SQL Reference*.

If you move a CICS application from OS/390 or VSE to run under another CICS product (for example, CICS for AIX), it can also access the OS/390 or VSE database using DB2 Connect. Refer to the *CICS/6000 Application Programming Guide* and the *CICS Customization and Operation* manual for more details.

**Note:** You can use DB2 Connect with a DB2 Universal Database Version 6 database, although it would be more efficient to use the DB2 private protocol without DB2 Connect. Most of the incompatibility issues listed in the following sections will not apply if you are using DB2 Connect against a DB2 Universal Database Version 6 database, except in cases where a restriction is due to a limitation of DB2 Connect itself, for example, the non-support of Abstract Data Types.

When you program in a host or AS/400 environment, you should consider the following specific factors:

- Using Data Definition Language (DDL)
- Using Data Manipulation Language (DML)
- Using Data Control Language (DCL)
- Connecting and disconnecting
- Precompiling
- Defining a sort order
- Managing referential integrity
- Locking
- Differences in SQLCODEs and SQLSTATEs
- Using system catalogs
- Isolation levels
- Stored procedures
- NOT ATOMIC compound SQL
- Distributed unit of work
- SQL statements supported or rejected by DB2 Connect.

---

## Using Data Definition Language (DDL)

DDL statements differ among the IBM database products because storage is handled differently on different systems. On host or AS/400 server systems, there can be several steps between designing a database and issuing a CREATE TABLE statement. For example, a series of statements may translate the design of logical objects into the physical representation of those objects in storage.

The precompiler passes many such DDL statements to the host or AS/400 server when you precompile to a host or AS/400 server database. The same statements would not precompile against a database on the system where the application is running. For example, in an OS/2 application the CREATE STORGROUP statement will precompile successfully to a DB2 Universal Database for OS/390 database, but not to a DB2 for OS/2 database.

---

## Using Data Manipulation Language (DML)

In general, DML statements are highly portable. SELECT, INSERT, UPDATE, and DELETE statements are similar across the IBM relational database products. Most applications primarily use DML SQL statements, which are supported by the DB2 Connect program.

### Numeric Data Types

When numeric data is transferred to DB2 Universal Database, the data type may change. Numeric and zoned decimal SQLTYPEs (supported by DB2 Universal Database for AS/400) are converted to fixed (packed) decimal SQLTYPEs.

### Mixed-Byte Data

Mixed-byte data can consist of characters from an extended UNIX code (EUC) character set, a double-byte character set (DBCS) and a single-byte character set (SBCS) in the same column. On systems that store data in EBCDIC (OS/390, OS/400, VSE, and VM), shift-out and shift-in characters mark the start and end of double-byte data. On systems that store data in ASCII (such as OS/2 and UNIX), shift-in and shift-out characters are not required.

If your application transfers mixed-byte data from an ASCII system to an EBCDIC system, be sure to allow enough room for the shift characters. For each switch from SBCS to DBCS data, add 2 bytes to your data length. For better portability, use variable-length strings in applications that use mixed-byte data.

## Long Fields

Long fields (strings longer than 254 characters) are handled differently on different systems. A host or AS/400 server may support only a subset of scalar functions for long fields; for example, DB2 Universal Database for OS/390 allows only the **LENGTH** and **SUBSTR** functions for long fields. Also, a host or AS/400 server may require different handling for certain SQL statements; for example, DB2 for VSE & VM requires that with the INSERT statement, only a host variable, the SQLDA, or a NULL value be used.

## Large Object (LOB) Data Type

The LOB data type is supported by DB2 Connect.

## User Defined Types (UDTs)

Only User Defined Distinct Types are supported by DB2 Connect. Abstract Data Types are not.

## ROWID Data Type

The ROWID data type is handled by DB2 Connect Version 6 as VARCHAR for bit data.

## 64-bit Integer (BIGINT) data type

Eight byte (64-bit) integers are supported by DB2 Connect Version 6. The BIGINT internal datatype is used in order to provide support for the cardinality of very large databases, while retaining data precision.

---

## Using Data Control Language (DCL)

Each IBM relational database management system provides different levels of granularity for the GRANT and REVOKE SQL statements. Check the product-specific publications to verify the appropriate SQL statements to use for each database management system.

---

## Connecting and Disconnecting

DB2 Connect supports the CONNECT TO and CONNECT RESET versions of the CONNECT statement, as well as CONNECT with no parameters. If an application calls an SQL statement without first performing an explicit CONNECT TO statement, an *implicit* connect is performed to the default application server (if one is defined).

When you connect to a database, information identifying the relational database management system is returned in the SQLERRP field of the SQLCA. If the application server is an IBM relational database, the first three bytes of SQLERRP contain one of the following:

**DSN** DB2 Universal Database for OS/390

**ARI** DB2 for VSE & VM

**QSQ** DB2 Universal Database for AS/400

**SQL** DB2 Universal Database.

If you issue a CONNECT TO or null CONNECT statement while using DB2 Connect, the country code or territory token in the SQLERRMC field of the SQLCA is returned as blanks; the CCSID of the application server is returned in the code page or code set token.

You can explicitly disconnect by using the CONNECT RESET statement (for type 1 connect), the RELEASE and COMMIT statements (for type 2 connect), or the DISCONNECT statement (either type of connect, but not in a TP monitor environment). Type 2 connect is used for distributed unit of work, as described in “Multisite Update” on page 434.

If a connection is not explicitly disconnected and the application ends normally, DB2 Connect commits the resulting data implicitly.

**Note:** An application can receive SQLCODEs indicating errors and still end normally; DB2 Connect commits the data in this case. If you do not want the data to be committed, you must issue a ROLLBACK command.

The FORCE command lets you disconnect selected users or all users from the database. This is supported for host or AS/400 server databases; the user can be forced off the DB2 Connect workstation.

---

## Precompiling

There are some differences in the precompilers for different IBM relational database systems. The precompiler for DB2 Universal Database differs from the host or AS/400 server precompilers in the following ways:

- It makes only one pass through an application.
- When binding against DB2 Universal Database databases, objects must exist for a successful bind. VALIDATE RUN is not supported.

## Blocking

The DB2 Connect program supports the DB2 database manager blocking bind options:

### **UNAMBIG**

Only unambiguous cursors are blocked (the default).

**ALL** Ambiguous cursors are blocked.

**NO** Cursors are not blocked.

The DB2 Connect program uses the block size defined in the DB2 database manager configuration file for the RQRIOBLK field. Current versions of DB2 Connect support block sizes up to 32 767. If larger values are specified in the DB2 database manager configuration file, DB2 Connect uses a value of 32 767 but does not reset the DB2 database manager configuration file. Blocking is handled the same way using the same block size for dynamic and static SQL.

**Note:** Most host or AS/400 server systems consider dynamic cursors ambiguous, but DB2 Universal Database systems consider some dynamic cursors unambiguous. To avoid confusion, you can specify **BLOCKING ALL** with DB2 Connect.

Specify the block size in the DB2 database manager configuration file by using the CLP, the Control Center, or an API, as listed in the *Administrative API Reference* and *Command Reference*.

## Package Attributes

A package has the following attributes:

### **Collection ID**

The ID of the package. It can be specified on the PREP command.

### **Owner**

The authorization ID of the package owner. It can be specified on the PREP or BIND command.

### **Creator**

The user name that binds the package.

### **Qualifier**

The implicit qualifier for objects in the package. It can be specified on the PREP or BIND command.

Each host or AS/400 server system has limitations on the use of these attributes:



### **DB2 Universal Database for OS/390**

All four attributes can be different. The use of a different qualifier requires special administrative privileges. For more information on the conditions concerning the usage of these attributes, refer to the *Command Reference* for DB2 Universal Database for OS/390.

### **DB2 for VSE & VM**

All of the attributes must be identical. If USER1 creates a bind file (with PREP), and USER2 performs the actual bind, USER2 needs DBA authority to bind for USER1. Only USER1's user name is used for the attributes.

### **DB2 Universal Database for AS/400**

The qualifier indicates the collection name. The relationship between qualifiers and ownership affects the granting and revoking of privileges on the object. The user name that is logged on is the creator and owner unless it is qualified by a collection ID, in which case the collection ID is the owner. The collection ID must already exist before it is used as a qualifier.

### **DB2 Universal Database**

All four attributes can be different. The use of a different owner requires administrative authority and the binder must have CREATEIN privilege on the schema (if it already exists).

**Note:** DB2 Connect provides support for the *SET CURRENT PACKAGESET* command for DB2 Universal Database for OS/390 and DB2 Universal Database.

## **C Null-terminated Strings**

The CNULREQD bind option overrides the handling of null-terminated strings that are specified using the LANGLEVEL option. See "Null-terminated Strings" on page 514 for a description of how null-terminated strings are handled when prepared with the LANGLEVEL option set to MIA or SAA1. By default, CNULREQD is set to YES. This causes null-terminated strings to be interpreted according to MIA standards. If connecting to a DB2 Universal Database for OS/390 server it is strongly recommended to set CNULREQD to YES. You need to bind applications coded to SAA1 standards (with respect to null-terminated strings) with the CNULREQD option set to NO. Otherwise, null-terminated strings will be interpreted according to MIA standards, even if they are prepared using LANGLEVEL set to SAA1.

## **Standalone SQLCODE and SQLSTATE**

Standalone SQLCODE and SQLSTATE variables, as defined in ISO/ANS SQL92, are supported through the LANGLEVEL SQL92E precompile option. An SQL0020W warning will be issued at precompile time, indicating that

LANGLEVEL is not supported. This warning applies only to the features listed under LANGLEVEL MIA in the *Command Reference*, which is a subset of LANGLEVEL SQL92E.

---

## Defining a Sort Order

The differences between EBCDIC and ASCII cause differences in sort orders in the various database products, and also affect ORDER BY and GROUP BY clauses. One way to minimize these differences is to create a user-defined collating sequence that mimics the EBCDIC sort order. You can specify a collating sequence only when you create a new database. For more information, refer to the *Administrative API Reference* and the *Command Reference*.

**Note:** Database tables can now be stored on DB2 Universal Database for OS/390 in ASCII format. This permits faster exchange of data between DB2 Connect and DB2 Universal Database for OS/390, and removes the need to provide field procedures which must otherwise be used to convert data and resequence it.

---

## Managing Referential Integrity

Different systems handle referential constraints differently:

### **DB2 Universal Database for OS/390**

An index must be created on a primary key before a foreign key can be created using the primary key. Tables can reference themselves.

### **DB2 for VSE & VM**

An index is automatically created for a foreign key. Tables cannot reference themselves.

### **DB2 Universal Database for AS/400**

An index is automatically created for a foreign key. Tables can reference themselves.

### **DB2 Universal Database**

For DB2 Universal Database databases, an index is automatically created for a unique constraint, including a primary key. Tables can reference themselves.

Other rules vary concerning levels of cascade.

---

## Locking

The way in which the database server performs locking can affect some applications. For example, applications designed around row-level locking and the isolation level of cursor stability are not directly portable to systems that perform page-level locking. Because of these underlying differences, applications may need to be adjusted.

The DB2 Universal Database for OS/390 and DB2 Universal Database products have the ability to time-out a lock and send an error return code to waiting applications.

---

## Differences in SQLCODEs and SQLSTATEs

Different IBM relational database products do not always produce the same SQLCODEs for similar errors. You can handle this problem in either of two ways:

- Use the SQLSTATE instead of the SQLCODE for a particular error.

SQLSTATEs have approximately the same meaning across the database products, and the products produce SQLSTATEs that correspond to the SQLCODEs.

- Map the SQLCODEs from one system to another system.

By default, DB2 Connect maps SQLCODEs and tokens from each IBM host or AS/400 server system to your DB2 Universal Database system. You can specify your own SQLCODE mapping file if you want to override the default mapping or you are using a database server that does not have SQLCODE mapping (a non-IBM database server). You can also turn off SQLCODE mapping.

---

## Using System Catalogs

The system catalogs vary across the IBM database products. Many differences can be masked by the use of views. For information, see the documentation for the database server that you are using.

The catalog functions in CLI get around this problem by presenting support of the same API and result sets for catalog queries across the DB2 family.

---

## Numeric Conversion Overflows on Retrieval Assignments

Numeric conversion overflows on retrieval assignments may be handled differently by different IBM relational database products. For example, consider fetching a float column into an integer host variable from DB2 Universal Database for OS/390 and from DB2 Universal Database. When converting the float value to an integer value, a conversion overflow may occur. By default, DB2 Universal Database for OS/390 will return a warning SQLCODE and a null value to the application. In contrast, DB2 Universal Database will return a conversion overflow error. It is recommended that applications avoid numeric conversion overflows on retrieval assignments by fetching into appropriately sized host variables.

---

## Isolation Levels

DB2 Connect accepts the following isolation levels when you prep or bind an application:

**RR**     Repeatable Read  
**RS**     Read Stability  
**CS**     Cursor Stability  
**UR**     Uncommitted Read  
**NC**     No Commit

The isolation levels are listed in order from most protection to least protection. If the host or AS/400 server does not support the isolation level that you specify, the next higher supported level is used.

Table 43 shows the result of each isolation level on each host or AS/400 application server.

Table 43. Isolation Levels

DB2 Connect	DB2 Universal Database for OS/390	DB2 for VSE & VM	DB2 Universal Database for AS/400	DB2 Universal Database
RR	RR	RR	note 1	RR
RS	note 2	RR	COMMIT(*ALL)	RS
CS	CS	CS	COMMIT(*CS)	CS
UR	note 3	CS	COMMIT(*CHG)	UR
NC	note 4	note 5	COMMIT(*NONE)	UR

Table 43. Isolation Levels (continued)

DB2 Connect	DB2 Universal Database for OS/390	DB2 for VSE & VM	DB2 Universal Database for AS/400	DB2 Universal Database
-------------	-----------------------------------	------------------	-----------------------------------	------------------------

**Note:**

1. There is no equivalent COMMIT option on DB2 Universal Database for AS/400 that matches RR. DB2 Universal Database for AS/400 supports RR by locking the whole table.
2. Results in RR for Version 3.1, and results in RS for Version 4.1 with APAR PN75407 or Version 5.1.
3. Results in CS for Version 3.1, and results in UR for Version 4.1 or Version 5.1.
4. Results in CS for Version 3.1, and results in UR for Version 4.1 with APAR PN60988 or Version 5.1.
5. Isolation level NC is not supported with DB2 for VSE & VM.

With DB2 Universal Database for AS/400, you can access an unjournalled table if an application is bound with an isolation level of UR and blocking set to ALL, or if the isolation level is set to NC.

## Stored Procedures

- Invocation

A client program can invoke a server program by issuing an SQL CALL statement. Each server works a little differently to the other servers in this case.

### OS/390

The schema name must be no more than 8 bytes long, the procedure name must be no more than 18 bytes long, and the stored procedure must be defined in the SYSIBM.SYSPROCEDURES catalog on the server.

### VSE or VM

The procedure name must not be more than 18 bytes long and must be defined in the SYSTEM.SYSROUTINES catalog on the server.

### OS/400

The procedure name must be an SQL identifier. You can also use the DECLARE PROCEDURE or CREATE PROCEDURE statements to specify the actual path name (the schema-name or collection-name) to locate the stored procedure.

All CALL statements to DB2 for AS/400 from REXX/SQL must be dynamically prepared and executed by the application as the CALL statement implemented in REXX/SQL maps to CALL USING DESCRIPTOR.

For the syntax of the SQL CALL statement, refer to the *SQL Reference*.

You can invoke the server program on DB2 Universal Database with the same parameter convention that server programs use on DB2 Universal Database for OS/390, DB2 Universal Database for AS/400, or DB2 for VSE & VM. For more information on invoking DB2 Universal Database stored procedures, see “Chapter 6. Stored Procedures” on page 185. For more information on the parameter convention on other platforms, refer to the DB2 product documentation for that platform.

All the SQL statements in a stored procedure are executed as part of the SQL unit of work started by the client SQL program.

- Do not pass indicator values with special meaning to or from stored procedures.

Between DB2 Universal Database, the systems pass whatever you put into the indicator variables. However, when using DB2 Connect, you can only pass 0, -1, and -128 in the indicator variables.

- You should define a parameter to return any errors or warning encountered by the server application.

A server program on DB2 Universal Database can update the SQLCA to return any error or warning, but a stored procedure on DB2 Universal Database for OS/390 or DB2 Universal Database for AS/400 has no such support. If you want to return an error code from your stored procedure, you must pass it as a parameter. The SQLCODE and SQLCA is only set by the server for system detected errors.

- DB2 for VSE & VM Version 6 or higher and DB2 Universal Database for OS/390 Version 5.1 or higher are the only host or AS/400 Application Servers that can return the result sets of stored procedures at this time.

---

## NOT ATOMIC Compound SQL

Compound SQL allows multiple SQL statements to be grouped into a single executable block. This may reduce network overhead and improve response time.

DB2 Connect supports NOT ATOMIC compound SQL. This means that processing of compound SQL continues following an error. (With ATOMIC compound SQL, which is not supported by DB2 Connect, an error would roll back the entire group of compound SQL.)

Statements will continue execution until terminated by the application server. In general, execution of the compound SQL statement will be stopped only in the case of serious errors.

NOT ATOMIC compound SQL can be used with all of the supported host or AS/400 application servers.

If multiple SQL errors occur, the SQLSTATes of the first seven failing statements are returned in the SQLERRMC field of the SQLCA with a message that multiple errors occurred. For more information, refer to the *SQL Reference*.

---

## Multisite Update with DB2 Connect

DB2 Connect allows you to perform a multisite update, also known as an update of multiple databases within a single distributed unit of work (DUOW). Whether you can use this capability depends on several factors:

- Your application program must be precompiled with the CONNECT 2 and SYNCPOINT TWOPHASE options.
- If you have SNA network connections, you can use two-phase commit support provided by the DB2 Syncpoint Manager (SPM) function of DB2 Connect Enterprise Edition Version 6 on AIX, OS/2, and Windows NT. This enables the following database servers to participate in a distributed unit of work:
  - DB2 Universal Database for AS/400 Version 3.1 or later
  - DB2 for MVS/ESA Version 3.1 or later
  - DB2 Universal Database for OS/390 Version 5.1 or later
  - DB2 for VM & VSE Version V5.1 or later.

The above is true for native DB2 Universal Database applications and applications coordinated by an external Transaction Processing Monitor such as IBM TXSeries, CICS for Open Systems, Encina Monitor, and Microsoft Transaction Server.

- If you have TCP/IP network connections then the following database servers can participate in a distributed unit of work:
  - DB2 Universal Database for OS/390 V5.1 or later.

If the application is controlled by a Transaction Processing Monitor such as IBM TXSeries, CICS for Open Systems, Encina Monitor, or Microsoft Transaction Server, then you must use the DB2 Syncpoint Manager. With DB2 Universal Database Version 6 (including DB2 Universal Database Enterprise Edition, DB2 Enterprise - Extended Edition, and DB2 Connect Enterprise Edition), the DB2 Syncpoint Manager has been enhanced to support TCP/IP connections.

If the application is a native DB2 UDB application, then the DB2 Syncpoint Manager is not required for two phase commit.

If a common DB2 Connect EE gateway is used by both native DB2 applications and TP Monitor Applications to access host data over TCP/IP connections then the DB2 Syncpoint Manager must be used.

**Note:** If a single DB2 Connect Enterprise Edition gateway will be used to access host data using both SNA and TCP/IP network protocols and two phase commit is required, then the DB2 Syncpoint Manager must be used. This is true for both DB2 applications and TP Monitor applications.

**Notes:**

1. DB2 Common Server Version 2.1 databases can be updated with two-phase commit in a unit of work *only* when DB2 Universal Database for OS/390 Version 5.1 is *not* the transaction manager database.
2. If the TM\_DATABASE for the transaction is DB2 Universal Database for OS/390 Version 5.1, then DB2 CS V2.1 databases participating in that transaction become *READ-ONLY* for the client application.
3. DB2 Universal Database Version 6 client applications can participate in distributed units of work with different levels of database server only if DB2 Universal Database for OS/390 Version 5.1 is the *TM\_DATABASE* for the transaction. For further information refer to either *DB2 Connect Enterprise Edition for OS/2 and Windows NT Quick Beginnings*, or *DB2 Connect Personal Edition Quick Beginnings*.
4. The database levels supported for two-phase commit transactions over TCP/IP depend on the DB2 client level, the TM\_DATABASE level, and the participant database levels.

We strongly recommend that *all* clients accessing DB2 Universal Database Version 6 and DB2 Universal Database for OS/390 Version 5 databases be at DB2 Universal Database Version 6. DB2 Version 2.1 clients *cannot* initiate two-phase commit transactions if DB2 Universal Database for OS/390 Version 5 databases participate in the transaction.

---

## Host or AS/400 Server SQL Statements Supported by DB2 Connect

The following statements compile successfully for host or AS/400 server processing but not for processing with DB2 Universal Database systems:

- ACQUIRE
- DECLARE (modifier.(qualifier.)table\_name TABLE ...
- LABEL ON

These statements are also supported by the command line processor.

The following statements are supported for host or AS/400 server processing but are not added to the bind file or the package and are not supported by the command line processor:

- DESCRIBE statement\_name INTO descriptor\_name USING NAMES



- PREPARE statement\_name INTO descriptor\_name USING NAMES FROM ...

The precompiler makes the following assumptions:

- Host variables are input variables
- The statement is assigned a unique section number.

---

### Host or AS/400 Server SQL Statements Rejected by DB2 Connect

The following SQL statements are not supported by DB2 Connect and not supported by the command line processor:

- COMMIT WORK RELEASE
- DECLARE state\_name, statement\_name STATEMENT
- DESCRIBE statement\_name INTO descriptor\_name USING xxxx (where xxxx is ANY, BOTH, or LABELS)
- PREPARE statement\_name INTO descriptor\_name USING xxxx FROM :host\_variable (where xxxx is ANY, BOTH, or LABELS)
- PUT ...
- ROLLBACK WORK RELEASE
- SET :host\_variable = CURRENT ...

DB2 for VSE & VM extended dynamic SQL statements are rejected with -104 and syntax error SQLCODEs.



## Appendix E. Country Code and Code Page Support

Table 44 shows the languages and code sets supported by the Database Servers and how these values are mapped to country code and code page values that are used by the database manager.

The following is an explanation of each column of the table:

1. **Code Page** shows the IBM-defined code page as mapped from the operating system code set.
2. **Group** shows whether a code page is single-byte ("S") or multi-byte ("D"). The "-n" is a number used to create a letter-number combination. Matching combinations show where connection and conversion is allowed by DB2. For example, all "S-1" groups can work together.
3. **Code Set** shows the code set associated with the supported language. The code set is mapped to the DB2 Code Page.
4. **Tr.** shows the two letter territory identifier.
5. **Country Code** shows the country code that is used by the database manager internally for providing country-specific support.
6. **Locale** shows the locale values supported by the database manager.
7. **OS** shows the operating system that supports the languages and code sets.
8. **Country Name** shows the name of the country or countries.

Table 44. Supported Languages and Code Sets

Code Page	Group	Code-Set	Tr.	Country		OS	Country Name
----	-----	-----	--	---	-----	----	-----
437	S-1	IBM-437	AL	355	-	OS2	Albania
850	S-1	IBM-850	AL	355	-	OS2	Albania
819	S-1	ISO8859-1	AL	355	sq_AL	AIX	Albania
850	S-1	IBM-850	AL	355	Sq_AL	AIX	Albania
819	S-1	iso88591	AL	355	-	HP	Albania
1051	S-1	roman8	AL	355	-	HP	Albania
819	S-1	ISO8859-1	AL	355	-	Sun	Albania
1252	S-1	1252	AL	355	-	WIN	Albania
1275	S-1	1275	AL	355	-	Mac	Albania
37	S-1	IBM-037	AL	355	-	HOST	Albania
1140	S-1	IBM-1140	AL	355	-	HOST	Albania

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country		OS	Country Name
----	-----	-----	--	---	-----	----	-----
864	S-6	IBM-864	AA	785	-	OS2	Arabic Countries
1046	S-6	IBM-1046	AA	785	Ar_AA	AIX	Arabic Countries
1089	S-6	ISO8859-6	AA	785	ar_AA	AIX	Arabic Countries
1089	S-6	iso88596	AA	785	ar_SA.iso88596	HP	Arabic Countries
1256	S-6	1256	AA	785	-	WIN	Arabic Countries
420	S-6	IBM-420	AA	785	-	HOST	Arabic Countries
437	S-1	IBM-437	AU	61	-	OS2	Australia
850	S-1	IBM-850	AU	61	-	OS2	Australia
819	S-1	ISO8859-1	AU	61	en_AU	AIX	Australia
850	S-1	IBM-850	AU	61	En_AU	AIX	Australia
819	S-1	iso88591	AU	61	-	HP	Australia
1051	S-1	roman8	AU	61	-	HP	Australia
819	S-1	ISO8859-1	AU	61	en_AU	Sun	Australia
819	S-1	ISO8859-1	AU	61	en_AU	SCO	Australia
1252	S-1	1252	AU	61	-	WIN	Australia
1275	S-1	1275	AU	61	-	Mac	Australia
37	S-1	IBM-037	AU	61	-	HOST	Australia
1140	S-1	IBM-1140	AU	61	-	HOST	Australia
437	S-1	IBM-437	AT	43	-	OS2	Austria
850	S-1	IBM-850	AT	43	-	OS2	Austria
819	S-1	ISO8859-1	AT	43	ge_AT	AIX	Austria
850	S-1	IBM-850	AT	43	Ge_AT	AIX	Austria
819	S-1	iso88591	AT	43	-	HP	Austria
1051	S-1	roman8	AT	43	-	HP	Austria
819	S-1	ISO8859-1	AT	43	de_AT	SCO	Austria
819	S-1	ISO8859-1	AT	43	de_AT	Sun	Austria
1252	S-1	1252	AT	43	-	WIN	Austria
1275	S-1	1275	AT	43	-	Mac	Austria
37	S-1	IBM-037	AT	43	-	HOST	Austria
1140	S-1	IBM-1140	AT	43	-	HOST	Austria
915	S-11	ISO8859-5	BY	375	-	OS2	Belarus
915	S-11	ISO8859-5	BY	375	be_BY	AIX	Belarus
1131	S-11	IBM-1131	BY	375	-	OS2	Belarus
1251	S-11	1251	BY	375	-	WIN	Belarus
1283	S-11	1283	BY	375	-	Mac	Belarus
1025	S-11	IBM-1025	BY	375	-	HOST	Belarus

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	-----	-----
437	S-1	IBM-437	BE	32	-	OS2	Belgium
850	S-1	IBM-850	BE	32	-	OS2	Belgium
819	S-1	ISO8859-1	BE	32	nl_BE	AIX	Belgium
					fr_BE		
850	S-1	IBM-850	BE	32	Nl_BE	AIX	Belgium
					Fr_BE		
819	S-1	iso88591	BE	32	-	HP	Belgium
819	S-1	ISO8859-1	BE	32	fr_BE	SCO	Belgium
819	S-1	ISO8859-1	BE	32	nl_BE	SCO	Belgium
819	S-1	ISO8859-1	BE	32	nl_BE	Sun	Belgium
					fr_BE		
1252	S-1	1252	BE	32	-	WIN	Belgium
1275	S-1	1275	BE	32	-	Mac	Belgium
500	S-1	IBM-500	BE	32	-	HOST	Belgium
1148	S-1	IBM-1148	BE	32	-	HOST	Belgium
855	S-5	IBM-855	BG	359	-	OS2	Bulgaria
915	S-5	ISO8859-5	BG	359	-	OS2	Bulgaria
915	S-5	ISO8859-5	BG	359	bg_BG	AIX	Bulgaria
915	S-5	iso88595	BG	359	bg_BG.iso88595	HP	Bulgaria
1251	S-5	1251	BG	359	-	WIN	Bulgaria
1283	S-5	1283	BG	359	-	Mac	Bulgaria
1025	S-5	IBM-1025	BG	359	-	HOST	Bulgaria
850	S-1	IBM-850	BR	55	-	OS2	Brazil
850	S-1	IBM-850	BR	55	-	AIX	Brazil
819	S-1	ISO8859-1	BR	55	pt_BR	AIX	Brazil
819	S-1	ISO8859-1	BR	55	-	HP	Brazil
819	S-1	ISO8859-1	BR	55	pt_BR	SCO	Brazil
819	S-1	ISO8859-1	BR	55	pt_BR	Sun	Brazil
1252	S-1	1252	BR	55	-	WIN	Brazil
37	S-1	IBM-037	BR	55	-	HOST	Brazil
1140	S-1	IBM-1140	BR	55	-	HOST	Brazil
850	S-1	IBM-850	CA	1	-	OS2	Canada
850	S-1	IBM-850	CA	1	En_CA	AIX	Canada
819	S-1	ISO8859-1	CA	1	en_CA	AIX	Canada
819	S-1	iso88591	CA	1	fr_CA.iso88591	HP	Canada
1051	S-1	roman8	CA	1	fr_CA.roman8	HP	Canada
819	S-1	ISO8859-1	CA	1	en_CA	SCO	Canada
819	S-1	ISO8859-1	CA	1	fr_CA	SCO	Canada
819	S-1	ISO8859-1	CA	1	en_CA	Sun	Canada
819	S-1	ISO8859-1	CA	1	en_CA	Sun	Canada
1252	S-1	1252	CA	1	-	WIN	Canada
1275	S-1	1275	CA	1	-	Mac	Canada
37	S-1	IBM-037	CA	1	-	HOST	Canada
1140	S-1	IBM-1140	CA	1	-	HOST	Canada
863	S-1	IBM-863	CA	2	-	OS2	Canada (French)

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	----	-----
1381	D-4	IBM-1381	CN	86	-	OS2	China (PRC)
1386	D-4	GBK	CN	86	-	OS2	China (PRC)
1383	D-4	IBM-eucCN	CN	86	zh_CN	AIX	China (PRC)
1386	D-4	GBK	CN	86	Zh_CN.GBK	AIX	China (PRC)
1383	D-4	hp15CN	CN	86	zh_CN.hp15CN	HP	China (PRC)
1383	D-4	eucCN	CN	86	zh_CN	SCO	China (PRC)
1383	D-4	eucCN	CN	86	zh_CN.eucCN	SCO	China (PRC)
1383	D-4	gb2312	CN	86	zh chinese	Sun	China (PRC)
1381	D-4	IBM-1381	CN	86	-	WIN	China (PRC)
1386	D-4	GBK	CN	86	-	WIN	China (PRC)
935	D-4	IBM-935	CN	86	-	HOST	China (PRC)
1388	D-4	IBM-1388	CN	86	-	HOST	China (PRC)
852	S-2	IBM-852	HR	385	-	OS2	Croatia
912	S-2	ISO8859-2	HR	385	hr_HR	AIX	Croatia
912	S-2	iso88592	HR	385	hr_HR.iso88592	HP	Croatia
912	S-2	ISO8859-2	HR	385	hr_HR.ISO8859-2	SCO	Croatia
1250	S-2	1250	HR	385	-	WIN	Croatia
1282	S-2	1282	HR	385	-	Mac	Croatia
870	S-2	IBM-870	HR	385	-	HOST	Croatia
852	S-2	IBM-852	CZ	42	-	OS2	Czech Republic
912	S-2	ISO8859-2	CZ	42	cs_CZ	AIX	Czech Republic
912	S-2	iso88592	CZ	42	cs_CZ.iso88592	HP	Czech Republic
912	S-2	ISO8859-2	CZ	42	cs_CZ.ISO8859-2	SCO	Czech Republic
1250	S-2	1250	CZ	42	-	WIN	Czech Republic
1282	S-2	1282	CZ	42	-	Mac	Czech Republic
870	S-2	IBM-870	CZ	42	-	HOST	Czech Republic
850	S-1	IBM-850	DK	45	-	OS2	Denmark
819	S-1	ISO8859-1	DK	45	da_DK	AIX	Denmark
850	S-1	IBM-850	DK	45	Da_DK	AIX	Denmark
819	S-1	iso88591	DK	45	da_DK.iso88591	HP	Denmark
1051	S-1	roman8	DK	45	da_DK.roman8	HP	Denmark
819	S-1	ISO8859-1	DK	45	da	SCO	Denmark
819	S-1	ISO8859-1	DK	45	da_DA	SCO	Denmark
819	S-1	ISO8859-1	DK	45	da_DK	SCO	Denmark
819	S-1	ISO8859-1	DK	45	da	Sun	Denmark
819	S-1	ISO8859-1	DK	45	da	Sun	Denmark
1252	S-1	1252	DK	45	-	WIN	Denmark
1275	S-1	1275	DK	45	-	Mac	Denmark
277	S-1	IBM-277	DK	45	-	HOST	Denmark
1142	S-1	IBM-1142	DK	45	-	HOST	Denmark

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	-----	-----
922	S-10	IBM-922	EE	372	-	OS2	Estonia
922	S-10	IBM-922	EE	372	Et_EE	AIX	Estonia
922	S-10	IBM-922	EE	372	-	WIN	Estonia
1122	S-10	IBM-1122	EE	372	-	HOST	Estonia
437	S-1	IBM-437	FI	358	-	OS2	Finland
850	S-1	IBM-850	FI	358	-	OS2	Finland
819	S-1	ISO8859-1	FI	358	fi_FI	AIX	Finland
850	S-1	IBM-850	FI	358	Fi_FI	AIX	Finland
819	S-1	iso88591	FI	358	fi_FI.iso88591	HP	Finland
819	S-1	ISO8859-1	FI	358	fi	SCO	Finland
819	S-1	ISO8859-1	FI	358	fi_FI	SCO	Finland
819	S-1	ISO8859-1	FI	358	sv_FI	SCO	Finland
819	S-1	ISO8859-1	FI	358	-	Sun	Finland
1051	S-1	roman8	FI	358	-	HP	Finland
1252	S-1	1252	FI	358	-	WIN	Finland
1275	S-1	1275	FI	358	-	Mac	Finland
278	S-1	IBM-278	FI	358	-	HOST	Finland
1143	S-1	IBM-1143	FI	358	-	HOST	Finland
855	S-5	IBM-855	MK	389	-	OS2	FYR Macedonia
915	S-5	ISO8859-5	MK	389	-	OS2	FYR Macedonia
915	S-5	ISO8859-5	MK	389	mk_MK	AIX	FYR Macedonia
915	S-5	iso88595	MK	389	-	HP	FYR Macedonia
1251	S-5	1251	MK	389	-	WIN	FYR Macedonia
1283	S-5	1283	MK	389	-	Mac	FYR Macedonia
1025	S-5	IBM-1025	MK	389	-	HOST	FYR Macedonia
437	S-1	IBM-437	FR	33	-	OS2	France
850	S-1	IBM-850	FR	33	-	OS2	France
819	S-1	ISO8859-1	FR	33	fr_FR	AIX	France
850	S-1	IBM-850	FR	33	Fr_FR	AIX	France
819	S-1	iso88591	FR	33	fr_FR.iso88591	HP	France
1051	S-1	roman8	FR	33	fr_FR.roman8	HP	France
819	S-1	ISO8859-1	FR	33	fr	Sun	France
819	S-1	ISO8859-1	FR	33	fr	SCO	France
819	S-1	ISO8859-1	FR	33	fr_FR	SCO	France
1252	S-1	1252	FR	33	-	WIN	France
1275	S-1	1275	FR	33	-	Mac	France
297	S-1	IBM-297	FR	33	-	HOST	France
1147	S-1	IBM-1147	FR	33	-	HOST	France

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	----	-----
437	S-1	IBM-437	DE	49	-	OS2	Germany
850	S-1	IBM-850	DE	49	-	OS2	Germany
819	S-1	ISO8859-1	DE	49	de_DE	AIX	Germany
850	S-1	IBM-850	DE	49	De_DE	AIX	Germany
819	S-1	iso88591	DE	49	de_DE.iso88591	HP	Germany
1051	S-1	roman8	DE	49	de_DE.roman8	HP	Germany
819	S-1	ISO8859-1	DE	49	de	SCO	Germany
819	S-1	ISO8859-1	DE	49	de_DE	SCO	Germany
819	S-1	ISO8859-1	DE	49	de	Sun	Germany
1252	S-1	1252	DE	49	-	WIN	Germany
1275	S-1	1275	DE	49	-	Mac	Germany
273	S-1	IBM-273	DE	49	-	HOST	Germany
1141	S-1	IBM-1141	DE	49	-	HOST	Germany
819	S-1	ISO8859-1	DE	49	De_DE.88591	SINIX	Germany
819	S-1	ISO8859-1	DE	49	De_DE.6937	SINIX	Germany
NOTE: DB2 supports as ISO 8859-1; it should be ISO 6937							
813	S-7	ISO8859-7	GR	30	-	OS2	Greece
869	S-7	IBM-869	GR	30	-	OS2	Greece
813	S-7	ISO8859-7	GR	30	el_GR	AIX	Greece
813	S-7	iso88597	GR	30	el_GR.iso88597	HP	Greece
813	S-7	ISO8859-7	GR	30	el_GR.ISO8859-7	SCO	Greece
737	S-7	737	GR	30	-	WIN	Greece
1253	S-7	1253	GR	30	-	WIN	Greece
1280	S-7	1280	GR	30	-	Mac	Greece
423	S-7	IBM-423	GR	30	-	HOST	Greece
875	S-7	IBM-875	GR	30	-	HOST	Greece
852	S-2	IBM-852	HU	36	-	OS2	Hungary
912	S-2	ISO8859-2	HU	36	hu_HU	AIX	Hungary
912	S-2	iso88592	HU	36	hu_HU.iso88592	HP	Hungary
912	S-2	ISO8859-2	HU	36	hu_HU.ISO8859-2	SCO	Hungary
1250	S-2	1250	HU	36	-	WIN	Hungary
1282	S-2	1282	HU	36	-	Mac	Hungary
870	S-2	IBM-870	HU	36	-	HOST	Hungary
850	S-1	IBM-850	IS	354	-	OS2	Iceland
819	S-1	ISO8859-1	IS	354	is_IS	AIX	Iceland
850	S-1	IBM-850	IS	354	Is_IS	AIX	Iceland
819	S-1	iso88591	IS	354	is_IS.iso88591	HP	Iceland
1051	S-1	roman8	IS	354	is_IS.roman8	HP	Iceland
819	S-1	ISO8859-1	IS	354	is	SCO	Iceland
819	S-1	ISO8859-1	IS	354	is_IS	SCO	Iceland
819	S-1	ISO8859-1	IS	354	-	Sun	Iceland
1252	S-1	1252	IS	354	-	WIN	Iceland
1275	S-1	1275	IS	354	-	Mac	Iceland
871	S-1	IBM-871	IS	354	-	HOST	Iceland
1149	S-1	IBM-1149	IS	354	-	HOST	Iceland



Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	-----	-----
437	S-1	IBM-437	IE	353	-	OS2	Ireland
850	S-1	IBM-850	IE	353	-	OS2	Ireland
819	S-1	ISO8859-1	IE	353	en_IE	AIX	Ireland
850	S-1	IBM-850	IE	353	En_IE	AIX	Ireland
819	S-1	iso88591	IE	353	-	HP	Ireland
1051	S-1	roman8	IE	353	-	HP	Ireland
819	S-1	ISO8859-1	IE	353	en_IE	Sun	Ireland
819	S-1	ISO8859-1	IE	353	en_IE.ISO8859-1	SCO	Ireland
1252	S-1	1252	IE	353	-	WIN	Ireland
1275	S-1	1275	IE	353	-	Mac	Ireland
285	S-1	IBM-285	IE	353	-	HOST	Ireland
1146	S-1	IBM-1146	IE	353	-	HOST	Ireland
862	S-8	IBM-862	IL	972	-	OS2	Israel
916	S-8	ISO8859-8	IL	972	iw_IL	AIX	Israel
1255	S-8	1255	IL	972	-	WIN	Israel
424	S-8	IBM-424	IL	972	-	HOST	Israel
437	S-1	IBM-437	IT	39	-	OS2	Italy
850	S-1	IBM-850	IT	39	-	OS2	Italy
819	S-1	ISO8859-1	IT	39	it_IT	AIX	Italy
850	S-1	IBM-850	IT	39	It_IT	AIX	Italy
819	S-1	iso88591	IT	39	it_IT.iso88591	HP	Italy
1051	S-1	roman8	IT	39	it_IT.roman8	HP	Italy
819	S-1	ISO8859-1	IT	39	it	SCO	Italy
819	S-1	ISO8859-1	IT	39	it_IT	SCO	Italy
819	S-1	ISO8859-1	IT	39	it	Sun	Italy
1252	S-1	1252	IT	39	-	WIN	Italy
1275	S-1	1275	IT	39	-	Mac	Italy
280	S-1	IBM-280	IT	39	-	HOST	Italy
1144	S-1	IBM-1144	IT	39	-	HOST	Italy

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country		OS	Country Name
----	-----	-----	--	---	-----	----	-----
932	D-1	IBM-932	JP	81	-	OS2	Japan
942	D-1	IBM-942	JP	81	-	OS2	Japan
943	D-1	IBM-943	JP	81	-	OS2	Japan
954	D-1	IBM-eucJP	JP	81	ja_JP	AIX	Japan
932	D-1	IBM-932	JP	81	Ja_JP	AIX	Japan
954	D-1	eucJP	JP	81	ja_JP.eucJP	HP	Japan
5039	D-1	SJIS	JP	81	ja_JP.SJIS	HP	Japan
954	D-1	eucJP	JP	81	ja_	SCO	Japan
954	D-1	eucJP	JP	81	ja_JP	SCO	Japan
954	D-1	eucJP	JP	81	ja_JP.EUC	SCO	Japan
954	D-1	eucJP	JP	81	ja_JP.eucJP	SCO	Japan
954	D-1	eucJP	JP	81	ja japanese	Sun	Japan
943	D-1	IBM-943	JP	81	-	WIN	Japan
930	D-1	IBM-930	JP	81	-	HOST	Japan
939	D-1	IBM-939	JP	81	-	HOST	Japan
5026	D-1	IBM-5026	JP	81	-	HOST	Japan
5035	D-1	IBM-5035	JP	81	-	HOST	Japan
949	D-3	IBM-949	KR	82	-	OS2	Korea, South
970	D-3	IBM-eucKR	KR	82	ko_KR	AIX	Korea, South
970	D-3	eucKR	KR	82	ko_KR.eucKR	HP	Korea, South
970	D-3	eucKR	KR	82	ko_KR.eucKR	SGI	Korea, South
970	D-3	5601	KR	82	ko korean	Sun	Korea, South
1363	D-3	1363	KR	82	-	WIN	Korea, South
933	D-3	IBM-933	KR	82	-	HOST	Korea, South
1364	D-3	IBM-1364	KR	82	-	HOST	Korea, South
437	S-1	IBM-437	Lat	3	-	OS2	Latin America
850	S-1	IBM-850	Lat	3	-	OS2	Latin America
819	S-1	ISO8859-1	Lat	3	-	AIX	Latin America
850	S-1	IBM-850	Lat	3	-	AIX	Latin America
819	S-1	iso88591	Lat	3	-	HP	Latin America
819	S-1	ISO8859-1	Lat	3	-	Sun	Latin America
1051	S-1	roman8	Lat	3	-	HP	Latin America
1252	S-1	1252	Lat	3	-	WIN	Latin America
1275	S-1	1275	Lat	3	-	Mac	Latin America
284	S-1	IBM-284	Lat	3	-	HOST	Latin America
1145	S-1	IBM-1145	Lat	3	-	HOST	Latin America
921	S-10	IBM-921	LV	371	-	OS2	Latvia
921	S-10	IBM-921	LV	371	Lv_LV	AIX	Latvia
921	S-10	IBM-921	LV	371	-	WIN	Latvia
1112	S-10	IBM-1112	LV	371	-	HOST	Latvia

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	-----	-----
921	S-10	IBM-921	LT	370	-	OS2	Lithuania
921	S-10	IBM-921	LT	370	Lt_LT	AIX	Lithuania
921	S-10	IBM-921	LV	370	-	WIN	Lithuania
1112	S-10	IBM-1112	LV	370	-	HOST	Lithuania
437	S-1	IBM-437	NL	31	-	OS2	Netherlands
850	S-1	IBM-850	NL	31	-	OS2	Netherlands
819	S-1	ISO8859-1	NL	31	nl_NL	AIX	Netherlands
850	S-1	IBM-850	NL	31	Nl_NL	AIX	Netherlands
819	S-1	iso88591	NL	31	nl_NL.iso88591	HP	Netherlands
1051	S-1	roman8	NL	31	nl_NL.roman8	HP	Netherlands
819	S-1	ISO8859-1	NL	31	nl	SCO	Netherlands
819	S-1	ISO8859-1	NL	31	nl_NL	SCO	Netherlands
819	S-1	ISO8859-1	NL	31	nl	Sun	Netherlands
1252	S-1	1252	NL	31	-	WIN	Netherlands
1275	S-1	1275	NL	31	-	Mac	Netherlands
37	S-1	IBM-037	NL	31	-	HOST	Netherlands
1140	S-1	IBM-1140	NL	31	-	HOST	Netherlands
850	S-1	IBM-850	NZ	64	-	OS2	New Zealand
850	S-1	IBM-850	NZ	64	En_NZ	AIX	New Zealand
819	S-1	ISO8859-1	NZ	64	en_NZ	AIX	New Zealand
819	S-1	ISO8859-1	NZ	64	-	HP	New Zealand
819	S-1	ISO8859-1	NZ	64	en_NZ	SCO	New Zealand
819	S-1	ISO8859-1	NZ	64	en_NZ	Sun	New Zealand
1252	S-1	1252	NZ	64	-	WIN	New Zealand
37	S-1	IBM-037	NZ	64	-	HOST	New Zealand
1140	S-1	IBM-1140	NZ	64	-	HOST	New Zealand
850	S-1	IBM-850	NO	47	-	OS2	Norway
819	S-1	ISO8859-1	NO	47	no_NO	AIX	Norway
850	S-1	IBM-850	NO	47	No_NO	AIX	Norway
819	S-1	iso88591	NO	47	no_NO.iso88591	HP	Norway
1051	S-1	roman8	NO	47	no_NO.roman8	HP	Norway
819	S-1	ISO8859-1	NO	47	no	SCO	Norway
819	S-1	ISO8859-1	NO	47	no_NO	SCO	Norway
819	S-1	ISO8859-1	NO	47	no	Sun	Norway
1252	S-1	1252	NO	47	-	WIN	Norway
1275	S-1	1275	NO	47	-	Mac	Norway
277	S-1	IBM-277	NO	47	-	HOST	Norway
1142	S-1	IBM-1142	NO	47	-	HOST	Norway

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	----	-----
852	S-2	IBM-852	PL	48	-	OS2	Poland
912	S-2	ISO8859-2	PL	48	pl_PL	AIX	Poland
912	S-2	iso88592	PL	48	pl_PL.iso88592	HP	Poland
912	S-2	ISO8859-2	PL	48	pl_PL.ISO8859-2	SCO	Poland
1250	S-2	1250	PL	48	-	WIN	Poland
1282	S-2	1282	PL	48	-	Mac	Poland
870	S-2	IBM-870	PL	48	-	HOST	Poland
860	S-1	IBM-860	PT	351	-	OS2	Portugal
850	S-1	IBM-850	PT	351	-	OS2	Portugal
819	S-1	ISO8859-1	PT	351	pt_PT	AIX	Portugal
850	S-1	IBM-850	PT	351	Pt_PT	AIX	Portugal
819	S-1	iso88591	PT	351	pt_PT.iso88591	HP	Portugal
1051	S-1	roman8	PT	351	pt_PT.roman8	HP	Portugal
819	S-1	ISO8859-1	PT	351	pt	SCO	Portugal
819	S-1	ISO8859-1	PT	351	pt_PT	SCO	Portugal
819	S-1	ISO8859-1	PT	351	pt	Sun	Portugal
1252	S-1	1252	PT	351	-	WIN	Portugal
1275	S-1	1275	PT	351	-	Mac	Portugal
37	S-1	IBM-037	PT	351	-	HOST	Portugal
1140	S-1	IBM-1140	PT	351	-	HOST	Portugal
852	S-2	IBM-852	RO	40	-	OS2	Romania
912	S-2	ISO8859-2	RO	40	ro_RO	AIX	Romania
912	S-2	iso88592	RO	40	ro_RO.iso88592	HP	Romania
912	S-2	ISO8859-2	RO	40	ro_RO.ISO8859-2	SCO	Romania
1250	S-2	1250	RO	40	-	WIN	Romania
1282	S-2	1282	RO	40	-	Mac	Romania
870	S-2	IBM-870	RO	40	-	HOST	Romania
866	S-5	IBM-866	RU	7	-	OS2	Russia
915	S-5	ISO8859-5	RU	7	-	OS2	Russia
915	S-5	ISO8859-5	RU	7	ru_RU	AIX	Russia
915	S-5	iso88595	RU	7	ru_RU.iso88595	HP	Russia
915	S-5	ISO8859-5	RU	7	ru_RU.ISO8859-5	SCO	Russia
1251	S-5	1251	RU	7	-	WIN	Russia
1283	S-5	1283	RU	7	-	Mac	Russia
1025	S-5	IBM-1025	RU	7	-	HOST	Russia
855	S-5	IBM-855	SP	381	-	OS2	Serbia/Montenegro
915	S-5	ISO8859-5	SP	381	-	OS2	Serbia/Montenegro
915	S-5	ISO8859-5	SP	381	sr_SP	AIX	Serbia/Montenegro
915	S-5	iso88595	SP	381	-	HP	Serbia/Montenegro
1251	S-5	1251	SP	381	-	WIN	Serbia/Montenegro
1283	S-5	1283	SP	381	-	Mac	Serbia/Montenegro
1025	S-5	IBM-1025	SP	381	-	HOST	Serbia/Montenegro

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	-----	-----
852	S-2	IBM-852	SK	938	-	OS2	Slovakia
912	S-2	ISO8859-2	SK	938	sk_SK	AIX	Slovakia
912	S-2	iso88592	SK	938	sk_SK.iso88592	HP	Slovakia
912	S-2	ISO8859-2	SK	938	sk_SK.ISO8859-2	SCO	Slovakia
1250	S-2	1250	SK	938	-	WIN	Slovakia
1282	S-2	1282	SK	938	-	Mac	Slovakia
870	S-2	IBM-870	SK	938	-	HOST	Slovakia
852	S-2	IBM-852	SI	386	-	OS2	Slovenia
912	S-2	ISO8859-2	SI	386	sl_SI	AIX	Slovenia
912	S-2	iso88592	SI	386	sl_SI.iso88592	HP	Slovenia
912	S-2	ISO8859-2	SI	386	sl_SI.ISO8859-2	SCO	Slovenia
1250	S-2	1250	SI	386	-	WIN	Slovenia
1282	S-2	1282	SI	386	-	Mac	Slovenia
870	S-2	IBM-870	SI	386	-	HOST	Slovenia
437	S-1	IBM-437	ZA	27	-	OS2	South Africa
850	S-1	IBM-850	ZA	27	-	OS2	South Africa
819	S-1	ISO8859-1	ZA	27	en_ZA	AIX	South Africa
850	S-1	IBM-850	ZA	27	En_ZA	AIX	South Africa
819	S-1	iso88591	ZA	27	-	HP	South Africa
1051	S-1	roman8	ZA	27	-	HP	South Africa
819	S-1	ISO8859-1	ZA	27	-	Sun	South Africa
819	S-1	ISO8859-1	ZA	27	en_ZA.ISO8859-1	SCO	South Africa
1252	S-1	1252	ZA	27	-	WIN	South Africa
1275	S-1	1275	ZA	27	-	Mac	South Africa
285	S-1	IBM-285	ZA	27	-	HOST	South Africa
1146	S-1	IBM-1146	ZA	27	-	HOST	South Africa
437	S-1	IBM-437	ES	34	-	OS2	Spain
850	S-1	IBM-850	ES	34	-	OS2	Spain
819	S-1	ISO8859-1	ES	34	es_ES	AIX	Spain
850	S-1	IBM-850	ES	34	ca_ES	AIX	Spain (Catalan)
819	S-1	iso88591	ES	34	Es_ES	AIX	Spain
1051	S-1	roman8	ES	34	Ca_ES	AIX	Spain (Calalan)
819	S-1	ISO8859-1	ES	34	es_ES.iso88591	HP	Spain
1051	S-1	roman8	ES	34	es_ES.roman8	HP	Spain
819	S-1	ISO8859-1	ES	34	es	Sun	Spain
819	S-1	ISO8859-1	ES	34	es	SCO	Spain
819	S-1	ISO8859-1	ES	34	es_ES	SCO	Spain
1252	S-1	1252	ES	34	-	WIN	Spain
1275	S-1	1275	ES	34	-	Mac	Spain
284	S-1	IBM-284	ES	34	-	HOST	Spain
1145	S-1	IBM-1145	ES	34	-	HOST	Spain

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	----	-----
<hr/>							
437	S-1	IBM-437	SE	46	-	OS2	Sweden
850	S-1	IBM-850	SE	46	-	OS2	Sweden
819	S-1	ISO8859-1	SE	46	sv_SE	AIX	Sweden
850	S-1	IBM-850	SE	46	Sv_SE	AIX	Sweden
819	S-1	iso88591	SE	46	sv_SE.iso88591	HP	Sweden
1051	S-1	roman8	SE	46	sv_SE.roman8	HP	Sweden
819	S-1	ISO8859-1	SE	46	sv	SCO	Sweden
819	S-1	ISO8859-1	SE	46	sv_SE	SCO	Sweden
819	S-1	ISO8859-1	SE	46	sv	Sun	Sweden
1252	S-1	1252	SE	46	-	WIN	Sweden
1275	S-1	1275	SE	46	-	Mac	Sweden
278	S-1	IBM-278	SE	46	-	HOST	Sweden
1143	S-1	IBM-1143	SE	46	-	HOST	Sweden
<hr/>							
437	S-1	IBM-437	CH	41	-	OS2	Switzerland
850	S-1	IBM-850	CH	41	-	OS2	Switzerland
819	S-1	ISO8859-1	CH	41	de_CH	AIX	Switzerland
850	S-1	IBM-850	CH	41	De_CH	AIX	Switzerland
819	S-1	iso88591	CH	41	-	HP	Switzerland
1051	S-1	roman8	CH	41	-	HP	Switzerland
819	S-1	ISO8859-1	CH	41	de_CH	SCO	Switzerland
819	S-1	ISO8859-1	CH	41	fr_CH	SCO	Switzerland
819	S-1	ISO8859-1	CH	41	it_CH	SCO	Switzerland
819	S-1	ISO8859-1	CH	41	de_CH	Sun	Switzerland
1252	S-1	1252	CH	41	-	WIN	Switzerland
1275	S-1	1275	CH	41	-	Mac	Switzerland
500	S-1	IBM-500	CH	41	-	HOST	Switzerland
1148	S-1	IBM-1148	CH	41	-	HOST	Switzerland
<hr/>							
938	D-2	IBM-938	TW	88	-	OS2	Taiwan
948	D-2	IBM-948	TW	88	-	OS2	Taiwan
950	D-2	big5	TW	88	-	OS2	Taiwan
950	D-2	big5	TW	88	Zh_TW	AIX	Taiwan
964	D-2	IBM-eucTW	TW	88	zh_TW	AIX	Taiwan
950	D-2	big5	TW	88	zh_TW.big5	HP	Taiwan
964	D-2	eucTW	TW	88	zh_TW.eucTW	HP	Taiwan
950	D-2	big5	TW	88	big5	Sun	Taiwan
					zh_TW.big5		
964	D-2	cns11643	TW	88	zh_TW tchinese	Sun	Taiwan
950	D-2	big5	TW	88	-	WIN	Taiwan
937	D-2	IBM-937	TW	88	-	HOST	Taiwan
<hr/>							
874	S-20	TIS620-1	TH	66	-	OS2	Thailand
874	S-20	TIS620-1	TH	66	Th_TH	AIX	Thailand
874	S-20	tis620	TH	66	th_TH.tis620	HP	Thailand
874	S-20	TIS620-1	TH	66	-	WIN	Thailand
838	S-20	IBM-838	TH	66	-	HOST	Thailand

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	-----	-----
857	S-9	IBM-857	TR	90	-	OS2	Turkey
920	S-9	ISO8859-9	TR	90	tr_TR	AIX	Turkey
920	S-9	iso88599	TR	90	tr_TR.iso88599	HP	Turkey
920	S-9	ISO8859-9	TR	90	tr_TR.ISO8859-9	SCO	Turkey
1254	S-9	1254	TR	90	-	WIN	Turkey
1281	S-9	1281	TR	90	-	Mac	Turkey
1026	S-9	IBM-1026	TR	90	-	HOST	Turkey
437	S-1	IBM-437	GB	44	-	OS2	U.K.
850	S-1	IBM-850	GB	44	-	OS2	U.K.
819	S-1	ISO8859-1	GB	44	en_GB	AIX	U.K.
850	S-1	IBM-850	GB	44	En_GB	AIX	U.K.
819	S-1	iso88591	GB	44	en_GB.iso88591	HP	U.K.
1051	S-1	roman8	GB	44	en_GB.roman8	HP	U.K.
819	S-1	ISO8859-1	GB	44	en_UK	Sun	U.K.
819	S-1	ISO8859-1	GB	44	en_GB	SCO	U.K.
819	S-1	ISO8859-1	GB	44	en	SCO	U.K.
1252	S-1	1252	GB	44	-	WIN	U.K.
1275	S-1	1275	GB	44	-	Mac	U.K.
285	S-1	IBM-285	GB	44	-	HOST	U.K.
1146	S-1	IBM-1146	GB	44	-	HOST	U.K.
819	S-1	88591	GB	44	En_GB.88591	SINIX	U.K.
819	S-1	ISO8859-1	GB	44	En_GB.6937	SINIX	U.K.
NOTE: DB2 supports as ISO 8859-1; it should be ISO 6937							
1125	S-12	IBM-1125	UA	380	-	OS2	Ukraine
1124	S-12	IBM-1124	UA	380	uk_UA	AIX	Ukraine
1251	S-12	1251	UA	380	-	WIN	Ukraine
1123	S-12	IBM-1123	UA	380	-	HOST	Ukraine
437	S-1	IBM-437	US	1	-	OS2	USA
850	S-1	IBM-850	US	1	-	OS2	USA
819	S-1	ISO8859-1	US	1	en_US	AIX	USA
850	S-1	IBM-850	US	1	En_US	AIX	USA
819	S-1	iso88591	US	1	en_US.iso88591	HP	USA
1051	S-1	roman8	US	1	en_US.roman8	HP	USA
819	S-1	ISO8859-1	US	1	en_US	Sun	USA
819	S-1	ISO8859-1	US	1	en_US	SGI	USA
819	S-1	ISO8859-1	US	1	en_US	SCO	USA
1252	S-1	1252	US	1	-	WIN	USA
1275	S-1	1275	US	1	-	Mac	USA
37	S-1	IBM-037	US	1	-	HOST	USA
1140	S-1	IBM-1140	US	1	-	HOST	USA
1163	S-13	IBM-1163	VN	84	-	OS2	Vietnam
1163	S-13	IBM-1163	VN	84	vi_VN	AIX	Vietnam
1258	S-13	1258	VN	84	-	WIN	Vietnam
1164	S-13	IBM-1164	VN	84	-	HOST	Vietnam

Table 44. Supported Languages and Code Sets (continued)

Code Page	Group	Code-Set	Tr.	Country Code	Locale	OS	Country Name
----	-----	-----	--	---	-----	----	-----

Note:

The following map to Arabic Countries (AA):

```

/* Arabic (Saudi Arabia) */
/* Arabic (Iraq) */
/* Arabic (Egypt) */
/* Arabic (Libya) */
/* Arabic (Algeria) */
/* Arabic (Morocco) */
/* Arabic (Tunisia) */
/* Arabic (Oman) */
/* Arabic (Yemen) */
/* Arabic (Syria) */
/* Arabic (Jordan) */
/* Arabic (Lebanon) */
/* Arabic (Kuwait) */
/* Arabic (United Arab Emirates) */
/* Arabic (Bahrain) */
/* Arabic (Qatar) */
```

The following map to English (US):

```

/* English (Jamaica) */
/* English (Carribean) */
```

The following map to Latin America (Lat):

```

/* Spanish (Mexican) */
/* Spanish (Guatemala) */
/* Spanish (Costa Rica) */
/* Spanish (Panama) */
/* Spanish (Dominican Republic) */
/* Spanish (Venezuela) */
/* Spanish (Colombia) */
/* Spanish (Peru) */
/* Spanish (Argentina) */
/* Spanish (Ecuador) */
/* Spanish (Chile) */
/* Spanish (Uruguay) */
/* Spanish (Paraguay) */
/* Spanish (Bolivia) */
```

**Note:** The Solaris code page 950 does not support the following characters in IBM 950:



Code Range	Description	Sun Big-5	IBM Big-5
C6A1-C8FE	Symbols	Reserved area	Symbols
F9D6-F9FE	ETen extension	Reserved area	ETen extension
F286-F9A0	IBM selected chars	Reserved area	IBM selected

**Note:** Euro-symbol support is provided with this version of DB2 UDB. Microsoft Windows ANSI code pages are modified according to the latest definition from Microsoft to include the Euro-symbol in position 0x80. This position was previously undefined. In addition, the definition of code page 850 has changed to replace the character Dotless i (found at position 0xD5) with the Euro-symbol. DB2 UDB uses the new definitions of these code pages as the default to provide Euro-symbol support. This implementation is the appropriate default for current DB2 UDB customers who require Euro-symbol support, and should not impact other customers. However, if you would like to continue to use the previous definition of these code pages, you may copy the following files:

- 12520850.cnv
- 08501252.cnv
- IBM00850.ucs
- IBM01252.ucs

from this directory

sqllib/conv/alt/

to this directory

sqllib/conv/

after installation is complete. You may want to backup the existing IBM01252.usc and IBM00850.ucs before copying the non-euro versions over them. After copying the files you will not have the euro currency symbol support from DB2 UDB.



---

## Appendix F. Simulating EBCDIC Binary Collation

With DB2, you can collate character strings according to a user-defined collating sequence. You can use this feature to simulate EBCDIC binary collation.

As an example of how to simulate EBCDIC collation, suppose you want to create an ASCII database with code page 850, but you also want the character strings to be collated as if the data actually resides in an EBCDIC database with code page 500. See Figure 14 on page 714 for the definition of code page 500, and Figure 15 on page 715 for the definition of code page 850.

Consider the relative collation of four characters in a EBCDIC code page 500 database, when they are collated in binary:

Character	Code Page 500 Code Point
'a'	X'81'
'b'	X'82'
'A'	X'C1'
'B'	X'C2'

The code page 500 binary collation sequence (the desired sequence) is:

'a' < 'b' < 'A' < 'B'

If you create the database with ASCII code page 850, binary collation would yield:

Character	Code Page 850 Code Point
'a'	X'61'
'b'	X'62'
'A'	X'41'
'B'	X'42'

The code page 850 binary collation (which is not the desired sequence) is:

'A' < 'B' < 'a' < 'b'

To achieve the desired collation, you need to create your database with a user-defined collating sequence. A sample collating sequence for just this purpose is supplied with DB2 in the `sqlc850a.h` include file. The content of `sqlc850a.h` is shown in Figure 13 on page 712.

```

#ifndef SQL_H_SQLE850A
#define SQL_H_SQLE850A

#ifdef __cplusplus
extern "C" {
#endif

unsigned char sqle_850_500[256] = {
0x00,0x01,0x02,0x03,0x37,0x2d,0x2e,0x2f,0x16,0x05,0x25,0x0b,0x0c,0x0d,0x0e,0x0f,
0x10,0x11,0x12,0x13,0x3c,0x3d,0x32,0x26,0x18,0x19,0x3f,0x27,0x1c,0x1d,0x1e,0x1f,
0x40,0x4f,0x7f,0x7b,0x5b,0x6c,0x50,0x7d,0x4d,0x5d,0x5c,0x4e,0x6b,0x60,0x4b,0x61,
0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0x7a,0x5e,0x4c,0x7e,0x6e,0x6f,
0x7c, 0xc1, 0xc2, 0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,
0xd7,0xd8,0xd9,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0x4a,0xe0,0x5a,0x5f,0x6d,
0x79, 0x81, 0x82, 0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x91,0x92,0x93,0x94,0x95,0x96,
0x97,0x98,0x99,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xc0,0xbb,0xd0,0xa1,0x07,
0x68,0xdc,0x51,0x42,0x43,0x44,0x47,0x48,0x52,0x53,0x54,0x57,0x56,0x58,0x63,0x67,
0x71,0x9c,0x9e,0xcb,0xcc,0xcd,0xdb,0xdd,0xdf,0xec,0xfc,0x70,0xb1,0x80,0xbf,0xff,
0x45,0x55,0xce,0xde,0x49,0x69,0x9a,0x9b,0xab,0xaf,0xba,0xb8,0xb7,0xaa,0x8a,0x8b,
0x2b,0x2c,0x09,0x21,0x28,0x65,0x62,0x64,0xb4,0x38,0x31,0x34,0x33,0xb0,0xb2,0x24,
0x22,0x17,0x29,0x06,0x20,0x2a,0x46,0x66,0x1a,0x35,0x08,0x39,0x36,0x30,0x3a,0x9f,
0x8c,0xac,0x72,0x73,0x74,0x0a,0x75,0x76,0x77,0x23,0x15,0x14,0x04,0x6a,0x78,0x3b,
0xee,0x59,0xeb,0xed,0xcf,0xef,0xa0,0x8e,0xae,0xfe,0xfb,0xfd,0x8d,0xad,0xbc,0xbe,
0xca,0x8f,0x1b,0xb9,0xb6,0xb5,0xe1,0x9d,0x90,0xbd,0xb3,0xda,0xfa,0xea,0x3e,0x41
};
#ifdef __cplusplus
}
#endif

#endif /* SQL_H_SQLE850A */

```

Figure 13. User-Defined Collating Sequence - sqle\_850\_500

To see how to achieve code page 500 binary collation on code page 850 characters, examine the sample collating sequence in sqle\_850\_500. For each code page 850 character, its weight in the collating sequence is simply its corresponding code point in code page 500.

For example, consider the letter 'a'. This letter is code point X'61' for code page 850 as shown in Figure 15 on page 715. In the array sqle\_850\_500, letter 'a' is assigned a weight of X'81' (that is, the 98th element in the array sqle\_850\_500).

Consider how the four characters collate when the database is created with the above sample user-defined collating sequence:

Character	Code Page 850 Code Point / Weight (from sqle_850_500)
'a'	X'61' / X'81'
'b'	X'62' / X'82'

'A'	X'41' / X'C1'
'B'	X'42' / X'C2'

The code page 850 user-defined collation by weight (the desired collation) is:

'a' < 'b' < 'A' < 'B'

In this example, you achieve the desired collation by specifying the correct weights to simulate the desired behavior.

Closely observing the actual collating sequence, notice that the sequence itself is merely a conversion table, where the source code page is the code page of the data base (850) and the target code page is the desired binary collating code page (500). Other sample collating sequences supplied by DB2 enable different conversions. If a conversion table that you require is not supplied with DB2, additional conversion tables can be obtained from the IBM publication, *Character Data Representation Architecture, Reference and Registry*, SC09-2190. You will find the additional conversion tables in a CD-ROM enclosed with this publication.

For more details on collating sequences, see "Collating Sequences" on page 394. Also see the CREATE DATABASE API described in the *Administrative API Reference* for a description of the collating sequences supplied with DB2, and for the listing of a sample program (db\_udcs.c) that demonstrates how to create a database with a user-defined collating sequence.

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
<b>-0</b>	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	μ SM170000	¢ SC040000	{ SM110000	}	\ SM070000	0 ND100000
<b>-1</b>	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
<b>-2</b>	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
<b>-3</b>	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
<b>-4</b>	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
<b>-5</b>	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
<b>-6</b>	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO10000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
<b>-7</b>	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
<b>-8</b>	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
<b>-9</b>	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
<b>-A</b>	[ SM060000	] SM080000	! SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	1 ND011000	2 ND021000	3 ND031000
<b>-B</b>	. SP110000	\$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	 SM130000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
<b>-C</b>	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Ð LD620000	– SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
<b>-D</b>	( SP060000	) SP070000	– SP090000	' SP050000	ý LY110000	¸ SD410000	Ý LY120000	¨ SD170000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
<b>-E</b>	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	' SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
<b>-F</b>	! SP020000	^ SD150000	? SP150000	" SP040000	± SA020000	Ɔ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 00500

Figure 14. Code Page 500

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		► (SP) SM590000	0 SP010000	@ ND100000	P SM050000	` LP020000	p SD130000	Ç LP010000	É LC420000	á LE120000	☐ LA110000	☐ SF140000	☐ SF020000	ø LD630000	Ó LO120000	(SHY) SP320000
-1	☺ SS000000	◄ SM630000	! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000	ü LU170000	æ LA510000	í LI110000	☐ SF150000	☐ SF070000	Đ LD620000	ß LS610000	± SA020000
-2	☹ SS010000	↕ SM760000	" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	é LE110000	Æ LA520000	ó LO110000	☐ SF160000	☐ SF060000	Ê LE160000	Ô LO160000	= SM100000
-3	♥ SS020000	!! SP330000	# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	â LA150000	ô LO150000	ú LU110000	☐ SF110000	☐ SF080000	Ë LE180000	Ò LO140000	¾ NF050000
-4	♦ SS030000	¶ SM250000	\$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	ä LA170000	ö LO170000	ñ LN190000	☐ SF090000	☐ SF100000	È LE140000	õ LO190000	¶ SM250000
-5	♣ SS040000	§ SM240000	% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	à LA130000	ò LO130000	Ñ LN200000	Á LA120000	☐ SF050000	ı LI610000	Õ LO200000	§ SM240000
-6	♠ SS050000	■ SM700000	& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	å LA270000	û LU150000	ª SM210000	Â LA160000	ã LA190000	Í LI120000	µ SM170000	÷ SA060000
-7	• SM570000	↕ SM770000	' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	ç LC410000	ù LU130000	º SM200000	À LA140000	Ã LA200000	Î LI160000	þ LT630000	¾ SD410000
-8	■ SM570001	↑ SM320000	( SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ê LE150000	ÿ LY170000	ı SP160000	© SM520000	☐ SF380000	İ LI180000	þ LT640000	° SM190000
-9	○ SM750000	↓ SM330000	) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	ë LE170000	Ö LO180000	® SM530000	☐ SF230000	☐ SF390000	☐ SF040000	Ú LU120000	¨ SD170000
-A	☐ SM750002	→ SM310000	* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	è LE130000	Ü LU180000	¬ SM660000	☐ SF240000	☐ SF400000	☐ SF010000	Û LU160000	· SD630000
-B	♂ SM280000	← SM300000	+ SA010000	; SP140000	K LK020000	[ SM060000	k LK010000	{ SM110000	ï LI170000	ø LO610000	½ NF010000	☐ SF250000	☐ SF410000	☐ SF610000	Ü LU140000	¹ ND011000
-C	♀ SM290000	⌞ SA420000	, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	 SM130000	î LI150000	£ SC020000	¼ NF040000	☐ SF260000	☐ SF420000	☐ SF570000	Ý LY110000	³ ND031000
-D	♪ SM930000	↔ SM780000	- SP100000	= SA040000	M LM020000	J SM080000	m LM010000	} SM140000	ì LI130000	Ø LO620000	ı SP030000	☐ SF430000	☐ SF430000	ı SM650000	Ý LY120000	² ND021000
-E	♪ SM910000	▲ SM600000	. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ä LA180000	× SA070000	« SP170000	¥ SC050000	☐ SF440000	İ LI140000	¯ SM150000	■ SM470000
-F	☀ SM690000	▼ SV040000	/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000	◻ SM790000	Å LA280000	f SC070000	» SP180000	☐ SF030000	☐ SC010000	☐ SF600000	' SD110000	(RSP) SP300000

Code Page 00850

Figure 15. Code Page 850





---

## Appendix G. How the DB2 Library Is Structured

The DB2 Universal Database library consists of SmartGuides, online help, books and sample programs in HTML format. This section describes the information that is provided, and how to access it.

To access product information online, you can use the Information Center. You can view task information, DB2 books, troubleshooting information, sample programs, and DB2 information on the Web. See “Accessing Information with the Information Center” on page 728 for details.

---

### Completing Tasks with SmartGuides

SmartGuides help you complete some administration tasks by taking you through each task one step at a time. SmartGuides are available through the Control Center and the Client Configuration Assistant. The following table lists the SmartGuides.

**Note:** Create Database, Index, and Configure Multisite Update SmartGuide are available for the partitioned database environment.

SmartGuide	Helps You to...	How to Access...
<b>Add Database</b>	Catalog a database on a client workstation.	From the Client Configuration Assistant, click <b>Add</b> .
<b>Back up Database</b>	Determine, create, and schedule a backup plan.	From the Control Center, click with the right mouse button on the database you want to back up and select <b>Backup-&gt;Database using SmartGuide</b> .
<b>Configure Multisite Update SmartGuide</b>	Perform a multi-site update, a distributed transaction, or a two-phase commit.	From the Control Center, click with the right mouse button on the <b>Database</b> icon and select <b>Multisite Update</b> .
<b>Create Database</b>	Create a database, and perform some basic configuration tasks.	From the Control Center, click with the right mouse button on the <b>Databases</b> icon and select <b>Create-&gt;Database using SmartGuide</b> .

SmartGuide	Helps You to...	How to Access...
<b>Create Table</b>	Select basic data types, and create a primary key for the table.	From the Control Center, click with the right mouse button on the <b>Tables</b> icon and select <b>Create-&gt;Table using SmartGuide</b> .
<b>Create Table Space</b>	Create a new table space.	From the Control Center, click with the right mouse button on the <b>Table spaces</b> icon and select <b>Create-&gt;Table space using SmartGuide</b> .
<b>Index</b>	Advise which indexes to create and drop for all your queries.	From the Control Center, click with the right mouse button on the <b>Index</b> icon and select <b>Create-&gt;Index using SmartGuide</b> .
<b>Performance Configuration</b>	Tune the performance of a database by updating configuration parameters to match your business requirements.	From the Control Center, click with the right mouse button on the database you want to tune and select <b>Configure using SmartGuide</b> .
<b>Restore Database</b>	Recover a database after a failure. It helps you understand which backup to use, and which logs to replay.	From the Control Center, click with the right mouse button on the database you want to restore and select <b>Restore-&gt;Database using SmartGuide</b> .

## Accessing Online Help

Online help is available with all DB2 components. The following table describes the various types of help. You can also access DB2 information through the Information Center. For information see “Accessing Information with the Information Center” on page 728.

Type of Help	Contents	How to Access...
<b>Command Help</b>	Explains the syntax of commands in the command line processor.	<p>From the command line processor in interactive mode, enter:</p> <p><i>? command</i></p> <p>where <i>command</i> is a keyword or the entire command.</p> <p>For example, <i>? catalog</i> displays help for all the CATALOG commands, while <i>? catalog database</i> displays help for the CATALOG DATABASE command.</p>

Type of Help	Contents	How to Access...
Control Center Help Client Configuration Assistant Help Event Analyzer Help Command Center Help	Explains the tasks you can perform in a window or notebook. The help includes prerequisite information you need to know, and describes how to use the window or notebook controls.	From a window or notebook, click the <b>Help</b> push button or press the F1 key.
Message Help	Describes the cause of a message, and any action you should take.	<p>From the command line processor in interactive mode, enter:</p> <p><i>? XXXnnnnnn</i></p> <p>where <i>XXXnnnnnn</i> is a valid message identifier.</p> <p>For example, <i>? SQL30081</i> displays help about the SQL30081 message.</p> <p>To view message help one screen at a time, enter:</p> <p><i>? XXXnnnnnn   more</i></p> <p>To save message help in a file, enter:</p> <p><i>? XXXnnnnnn &gt; filename.ext</i></p> <p>where <i>filename.ext</i> is the file where you want to save the message help.</p>
SQL Help	Explains the syntax of SQL statements.	<p>From the command line processor in interactive mode, enter:</p> <p><i>help statement</i></p> <p>where <i>statement</i> is an SQL statement.</p> <p>For example, <b>help</b> SELECT displays help about the SELECT statement.</p> <p><b>Note:</b> SQL help is not available on UNIX-based platforms.</p>

Type of Help	Contents	How to Access...
<b>SQLSTATE Help</b>	Explains SQL states and class codes.	<p>From the command line processor in interactive mode, enter:</p> <p><i>? sqlstate</i> or <i>? class-code</i></p> <p>where <i>sqlstate</i> is a valid five-digit SQL state and <i>class-code</i> is the first two digits of the SQL state.</p> <p>For example, <i>? 08003</i> displays help for the 08003 SQL state, while <i>? 08</i> displays help for the 08 class code.</p>

---

## DB2 Information – Hardcopy and Online

The table in this section lists the DB2 books. They are divided into two groups:

### **Cross-platform books**

These books contain the common DB2 information for all platforms.

### **Platform-specific books**

These books are for DB2 on a specific platform. For example, there are separate *Quick Beginnings* books for DB2 on OS/2, on Windows NT, and on the UNIX-based platforms.

### **Cross-platform sample programs in HTML**

These samples are the HTML version of the sample programs that are installed with the SDK. They are for informational purposes and do not replace the actual programs.

Most books are available in HTML and PostScript format, or you can choose to order a hardcopy from IBM. The exceptions are noted in the table.

On OS/2 and Windows platforms, HTML documentation files can be installed under the doc\html subdirectory. Depending on the language of your system, some files may be in that language, and the remainder are in English.

On UNIX platforms, you can install multiple language versions of the HTML documentation files under the doc/%L/html subdirectories. Any documentation that is not available in a national language is shown in English.

You can obtain DB2 books and access information in a variety of different ways:

<b>View</b>	See “Viewing Online Information” on page 727.
<b>Search</b>	See “Searching Online Information” on page 730.
<b>Print</b>	See “Printing the PostScript Books” on page 730.
<b>Order</b>	See “Ordering the Printed Books” on page 732.

Name	Description	Form Number File Name for Online Book	HTML Directory
<b>Cross-Platform Books</b>			
<i>Administration Guide</i>	<p><i>Administration Guide, Design and Implementation</i> contains information required to design, implement, and maintain a database. It also describes database access using the Control Center(whether local or in a client/server environment), auditing, database recovery, distributed database support, and high availability.</p> <p><i>Administration Guide, Performance</i> contains information that focuses on the database environment, such as application performance evaluation and tuning.</p> <p>You can order both volumes of the <i>Administration Guide</i> in the English language in North America using the form number SBOF-8922.</p>	<p>Volume 1 SC09-2839 db2d1x60</p> <p>Volume 2 SC09-2840 db2d2x60</p>	db2d0
<i>Administrative API Reference</i>	Describes the DB2 application programming interfaces (APIs) and data structures you can use to manage your databases. Explains how to call APIs from your applications.	SC09-2841 db2b0x60	db2b0
<i>Application Building Guide</i>	<p>Provides environment setup information and step-by-step instructions about how to compile, link, and run DB2 applications on Windows, OS/2, and UNIX-based platforms.</p> <p>This book combines the <i>Building Applications</i> books for the OS/2, Windows, and UNIX-based environments.</p>	SC09-2842 db2axx60	db2ax

Name	Description	Form Number File Name for Online Book	HTML Directory
<i>APPC, CPI-C and SNA Sense Codes</i>	Provides general information about APPC, CPI-C, and SNA sense codes that you may encounter when using DB2 Universal Database products. <b>Note:</b> Available in HTML format only.	No form number db2apx60	db2ap
<i>Application Development Guide</i>	Explains how to develop applications that access DB2 databases using embedded SQL or JDBC, how to write stored procedures, user-defined types, user-defined functions, and how to use triggers. It also discusses programming techniques and performance considerations.  This book was formerly known as the <i>Embedded SQL Programming Guide</i> .	SC09-2845 db2a0x60	db2a0
<i>CLI Guide and Reference</i>	Explains how to develop applications that access DB2 databases using the DB2 Call Level Interface, a callable SQL interface that is compatible with the Microsoft ODBC specification.	SC09-2843 db2l0x60	db2l0
<i>Command Reference</i>	Explains how to use the command line processor, and describes the DB2 commands you can use to manage your database.	SC09-2844 db2n0x60	db2n0
<i>Data Movement Utilities Guide and Reference</i>	Explains how to use the Load, Import, Export, Autoloader, and Data Propagation utilities to work with the data in the database.	SC09-2858 db2dmx60	db2dm
<i>DB2 Connect Personal Edition Quick Beginnings</i>	Provides planning, installing, and configuring information for DB2 Connect Personal Edition.	GC09-2830 db2c1x60	db2c1
<i>DB2 Connect User's Guide</i>	Provides concepts, programming and general usage information about the DB2 Connect products.	SC09-2838 db2c0x60	db2c0

Name	Description	Form Number File Name for Online Book	HTML Directory
<i>Connectivity Supplement</i>	Provides setup and reference information on how to use DB2 for AS/400, DB2 for OS/390, DB2 for MVS, or DB2 for VM as DRDA application requesters with DB2 Universal Database servers, and on how to use DRDA application servers with DB2 Connect application requesters. <b>Note:</b> Available in HTML and PostScript formats only.	No form number db2h1x60	db2h1
<i>Glossary</i>	Provides a comprehensive list of all DB2 terms and definitions. <b>Note:</b> Available in HTML format only.	No form number db2t0x50	db2t0
<i>Installation and Configuration Supplement</i>	Guides you through the planning, installation, and set up of platform-specific DB2 clients. This supplement contains information on binding, setting up client and server communications, DB2 GUI tools, DRDA AS, distributed installation, and the configuration of distributed requests and access methods to heterogeneous data sources.	GC09-2857 db2iyx60	db2iy
<i>Message Reference</i>	Lists messages and codes issued by DB2, and describes the actions you should take.	GC09-2846 db2m0x60	db2m0
<i>Replication Guide and Reference</i>	Provides planning, configuration, administration, and usage information for the IBM Replication tools supplied with DB2.	SC26-9642 db2e0x60	db2e0
<i>SQL Getting Started</i>	Introduces SQL concepts, and provides examples for many constructs and tasks.	SC09-2856 db2y0x60	db2y0
<i>SQL Reference, Volume 1 and Volume 2</i>	Describes SQL syntax, semantics, and the rules of the language. Also includes information about release-to-release incompatibilities, product limits, and catalog views.  You can order both volumes of the <i>SQL Reference</i> in the English language in North America with the form number SBOF-8923.	SBOF-8923  Volume 1 db2s1x60  Volume 2 db2s2x60	db2s0

Name	Description	Form Number File Name for Online Book	HTML Directory
<i>System Monitor Guide and Reference</i>	Describes how to collect different kinds of information about databases and the database manager. Explains how to use the information to understand database activity, improve performance, and determine the cause of problems.	SC09-2849 db2f0x60	db2f0
<i>Troubleshooting Guide</i>	Helps you determine the source of errors, recover from problems, and use diagnostic tools in consultation with DB2 Customer Service.	S10J-8169	db2p0
<i>What's New</i>	Describes the new features, functions, and enhancements in DB2 Universal Database, Version 6.0, including information about Java-based tools.	SC09-2851 db2q0x60	db2q0
<b>Platform-Specific Books</b>			
<i>Administering Satellites Guide and Reference</i>	Provides planning, configuration, administration, and usage information for satellites.	GC09-2821 db2dsx60	db2ds
<i>DB2 Personal Edition Quick Beginnings</i>	Provides planning, installation, migration, and configuration information for DB2 Universal Database Personal Edition on the OS/2, Windows 95, and Windows NT operating systems.	GC09-2831 db2i1x60	db2i1
<i>DB2 for OS/2 Quick Beginnings</i>	Provides planning, installation, migration, and configuration information for DB2 Universal Database on the OS/2 operating system. Also contains installing and setup information for many supported clients.	GC09-2834 db2i2x60	db2i2
<i>DB2 for UNIX Quick Beginnings</i>	Provides planning, installation, migration, and configuration information for DB2 Universal Database on UNIX-based platforms. Also contains installing and setup information for many supported clients.	GC09-2836 db2ixx60	db2ix
<i>DB2 for Windows NT Quick Beginnings</i>	Provides planning, installation, migration, and configuration information for DB2 Universal Database on the Windows NT operating system. Also contains installing and setup information for many supported clients.	GC09-2835 db2i6x60	db2i6



Name	Description	Form Number File Name for Online Book	HTML Directory
<i>DB2 Enterprise - Extended Edition for UNIX Quick Beginnings</i>	Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for UNIX. Also contains installing and setup information for many supported clients.	GC09-2832 db2v3x60	db2v3
<i>DB2 Enterprise - Extended Edition for Windows NT Quick Beginnings</i>	Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for Windows NT. Also contains installing and setup information for many supported clients.	GC09-2833 db2v6x60	db2v6
<i>DB2 Connect Enterprise Edition for OS/2 and Windows NT Quick Beginnings</i>	Provides planning, migration, installation, and configuration information for DB2 Connect Enterprise Edition on the OS/2 and Windows NT operating systems. Also contains installation and setup information for many supported clients.  This book was formerly part of the <i>DB2 Connect Enterprise Edition Quick Beginnings</i> .	GC09-2828 db2c6x60	db2c6
<i>DB2 Connect Enterprise Edition for UNIX Quick Beginnings</i>	Provides planning, migration, installation, configuration, and usage information for DB2 Connect Enterprise Edition in UNIX-based platforms. Also contains installation and setup information for many supported clients.  This book was formerly part of the <i>DB2 Connect Enterprise Edition Quick Beginnings</i> .	GC09-2829 db2cyx60	db2cy
<i>DB2 Data Links Manager for AIX Quick Beginnings</i>	Provides planning, installation, configuration, and task information for DB2 Data Links Manager for AIX.	GC09-2837 db2z0x60	db2z0
<i>DB2 Data Links Manager for Windows NT Quick Beginnings</i>	Provides planning, installation, configuration, and task information for DB2 Data Links Manager for Windows NT.	GC09-2827 db2z6x60	db2z6
<i>DB2 Query Patroller Administration Guide</i>	Provides administration information on DB2 Query Patrol.	SC09-2859 db2dwx60	db2dw

Name	Description	Form Number File Name for Online Book	HTML Directory
<i>DB2 Query Patroller Installation Guide</i>	Provides installation information on DB2 Query Patrol.	GC09-2860 db2iwx60	db2iw
<i>DB2 Query Patroller User's Guide</i>	Describes how to use the tools and functions of the DB2 Query Patrol.	SC09-2861 db2www60	db2ww
<b>Cross-Platform Sample Programs in HTML</b>			
Sample programs in HTML	Provides the sample programs in HTML format for the programming languages on all platforms supported by DB2 for informational purposes (not all samples are available in all languages). Only available when the SDK is installed.  See <i>Application Building Guide</i> for more information on the actual programs. <b>Note:</b> Available in HTML format only.	No form number	db2hs/c db2hs/cli db2hs/clp db2hs/cpp db2hs/cobol db2hs/cobol_mf db2hs/fortran db2hs/java db2hs/rexx

**Notes:**

1. The character in the sixth position of the file name indicates the language of a book. For example, the file name db2d0e60 indicates that the *Administration Guide* is in English. The following letters are used in the file names to indicate the language of a book:

Language	Identifier
Brazilian Portuguese	b
Bulgarian	u
Czech	x
Danish	d
Dutch	q
English	e
Finnish	y
French	f
German	g
Greek	a
Hungarian	h
Italian	i
Japanese	j
Korean	k
Norwegian	n
Polish	p

Portuguese	v
Russian	r
Simp. Chinese	c
Slovenian	l
Spanish	z
Swedish	s
Trad. Chinese	t
Turkish	m

2. For late breaking information that could not be included in the DB2 books:

- On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/%L directory, where %L is the locale name and DB2DIR is:
  - /usr/lpp/db2\_06\_01 on AIX
  - /opt/IBMDb2/V6.1 on HP-UX, Solaris, SCO UnixWare 7, and Silicon Graphics IRIX
  - /usr/IBMDb2/V6.1 on Linux.
- On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.
- Under Windows Start menu

---

## Viewing Online Information

The manuals included with this product are in Hypertext Markup Language (HTML) softcopy format. Softcopy format enables you to search or browse the information, and provides hypertext links to related information. It also makes it easier to share the library across your site.

You can view the online books or sample programs with any browser that conforms to HTML Version 3.2 specifications.

To view online books or sample programs on all platforms other than SCO UnixWare 7:

- If you are running DB2 administration tools, use the Information Center. See “Accessing Information with the Information Center” on page 728 for details.
- Select the Open Page menu item of your Web browser. The page you open contains descriptions of and links to DB2 information:
  - On UNIX-based platforms, open the following page:  
`file:/INSTHOME/sql1lib/doc/%L/html/index.htm`

where %L is the locale name.

- On other platforms, open the following page:

`sql1lib\doc\html\index.htm`

The path is located on the drive where DB2 is installed.

If you have not installed the Information Center, you can open the page by double-clicking on the **DB2 Online Books** icon. Depending on the system you are using, the icon is in the main product folder or the Windows Start menu.

To view online books or sample programs on the SCO UnixWare 7:

- DB2 Universal Database for SCO UnixWare 7 uses the native SCOhelp utility to search the DB2 information. You can access SCOhelp by the following methods:
  - entering the "scohelp" command on the command line,
  - selecting the Help menu in the Control Panel of the CDE desktop or
  - selecting Help in the Root menu of the Panorama desktop

For more information on SCOhelp, refer to the *Installation and Configuration Supplement*.

## Accessing Information with the Information Center

The Information Center provides quick access to DB2 product information. The Information Center is available on all platforms on which the DB2 administration tools are available.

Depending on your system, you can access the Information Center from the:

- Main product folder
- Toolbar in the Control Center
- Windows Start menu
- Help menu of the Control Center

The Information Center provides the following kinds of information. Click the appropriate tab to look at the information:

<b>Tasks</b>	Lists tasks you can perform using DB2.
<b>Reference</b>	Lists DB2 reference information, such as keywords, commands, and APIs.
<b>Books</b>	Lists DB2 books.
<b>Troubleshooting</b>	Lists categories of error messages and their recovery actions.
<b>Sample Programs</b>	Lists sample programs that come with the

DB2 Software Developer's Kit. If the Software Developer's Kit is not installed, this tab is not displayed.

## Web

Lists DB2 information on the World Wide Web. To access this information, you must have a connection to the Web from your system.

When you select an item in one of the lists, the Information Center launches a viewer to display the information. The viewer might be the system help viewer, an editor, or a Web browser, depending on the kind of information you select.

The Information Center provides some search capabilities, so you can look for specific topics, and filter capabilities to limit the scope of your searches.

For a full text search, click the Search button of the Information Center follow the **Search DB2 Books** link in each HTML file.

The HTML search server is usually started automatically. If a search in the HTML information does not work, you may have to start the search server by double-clicking its icon on the Windows or OS/2 desktop.

Refer to the release notes if you experience any other problems when searching the HTML information.

**Note:** Search function is not available in the Linux and Silicon Graphics environments.

---

## Setting Up a Document Server

By default, the DB2 information is installed on your local system. This means that each person who needs access to the DB2 information must install the same files. To have the DB2 information stored in a single location, use the following instructions:

1. Copy all files and subdirectories from `\sqllib\doc\html` on your local system to a Web server. Each book has its own subdirectory containing all the necessary HTML and GIF files that make up the book. Ensure that the directory structure remains the same.
2. Configure the Web server to look for the files in the new location. For information, see the NetQuestion Appendix in *Installation and Configuration Supplement*.

3. If you are using the Java version of the Information Center, you can specify a base URL for all HTML files. You should use the URL for the list of books.
4. Once you are able to view the book files, you should bookmark commonly viewed topics. Among those, you will probably want to bookmark the following pages:
  - List of books
  - Tables of contents of frequently used books
  - Frequently referenced articles, such as the **ALTER TABLE** topic
  - The Search form

For information about setting up a search, see the NetQuestion Appendix in *Installation and Configuration Supplement* book.

---

## Searching Online Information

To search for information in the HTML books, you can do the following:

- Click on **Search the DB2 Books** at the bottom of any page in the HTML books. Use the search form to find a specific topic. This function is not available in the Linux or Silicon Graphics IRIX environments.
- Click on **Index** at the bottom of any page in an HTML book. Use the index to find a specific topic in the book.
- Display the table of contents or index of the HTML book, and then use the find function of the Web browser to find a specific topic in the book.
- Use the bookmark function of the Web browser to quickly return to a specific topic.
- Use the search function of the Information Center to find specific topics. See “Accessing Information with the Information Center” on page 728 for details.

---

## Printing the PostScript Books

If you prefer to have printed copies of the manuals, you can decompress and print PostScript versions. For the file name of each book in the library, see the table in “DB2 Information – Hardcopy and Online” on page 720. Specify the full path name for the file you intend to print.

On OS/2 and Windows platforms:

1. Copy the compressed PostScript files to a hard drive on your system. The files have a file extension of .exe and are located in the `x:\doc\language\books\ps` directory, where x: is the letter representing the

CD-ROM drive and *language* is the two-character country code that represents your language (for example, EN for English).

2. Decompress the file that corresponds to the book that you want. Each compressed book is a self-extracting executable file. To decompress the book, simply run it as you would run any other executable program. The result from this step is a printable PostScript file with a file extension of .ps.
3. Ensure that your default printer is a PostScript printer capable of printing Level 1 (or equivalent) files.
4. Enter the following command from a command line:

```
print filename.ps
```

On UNIX-based platforms:

1. Mount the CD-ROM. Refer to your *Quick Beginnings* manual for the procedures to mount the CD-ROM.
2. Change to /cdrom/doc/%L/ps directory on the CD-ROM, where /cdrom is the mount point of the CD-ROM and %L is the name of the desired locale. The manuals will be installed in the previously-mentioned directory with file names ending with .ps.Z.
3. Decompress and print the manual you require using the following command:

- For AIX:

```
zcat filename | qprt -P PSprinter_queue
```

- For HP-UX, Solaris, or SCO UnixWare 7:

```
zcat filename | lp -d PSprinter_queue
```

- For Linux:

```
zcat filename | lpr -P PSprinter_queue
```

- For Silicon Graphics IRIX:

```
zcat < filename | lp -d PSprinter_queue
```

where *filename* is the full path name and extension of the compressed PostScript file and *PSprinter\_queue* is the name of the PostScript printer queue.

For example, to print the English version of *DB2 for UNIX Quick Beginnings* on AIX, you can use the following command:

```
zcat /cdrom/doc/en/ps/db2ixe60.ps.Z || qprt -P ps1
```

---

## Ordering the Printed Books

You can order the printed DB2 manuals either as a set or individually. There are three sets of books available. The form number for the entire set of DB2 books is SB0F-8926-00. The form number for the books listed under the heading "Cross-Platform Books" is SB0F-8924-00.

**Note:** These form numbers only apply if you are ordering books that are printed in the English language in North America.

You can also order books individually by the form number listed in "DB2 Information – Hardcopy and Online" on page 720. To order printed versions, contact your IBM authorized dealer or marketing representative, or phone 1-800-879-2755 in the United States or 1-800-IBM-4YOU in Canada.



---

## Appendix H. Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the

IBM Director of Licensing  
IBM Corporation, North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited  
Office of the Lab Director  
1150 Eglinton Ave. East  
North York, Ontario  
M3C 1H7  
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

---

## Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

ACF/VTAM	MVS/ESA
ADSTAR	MVS/XA
AISPO	OS/400
AIX	OS/390
AIXwindows	OS/2
AnyNet	PowerPC
APPN	QMF
AS/400	RACF
CICS	RISC System/6000
C Set++	SP
C/370	SQL/DS
DATABASE 2	SQL/400
DataHub	S/370
DataJoiner	System/370
DataPropagator	System/390
DataRefresher	SystemView
DB2	VisualAge
DB2 Connect	VM/ESA
DB2 Universal Database	VSE/ESA
Distributed Relational Database Architecture	VTAM
DRDA	WIN-OS/2
Extended Services	
FFST	
First Failure Support Technology	
IBM	
IMS	
LAN Distance	

---

## Trademarks of Other Companies

The following terms are trademarks or registered trademarks of the companies listed:

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

HP-UX is a trademark of Hewlett-Packard.

Java, HotJava, Solaris, Solstice, and Sun are trademarks of Sun Microsystems, Inc.

Microsoft, Windows, Windows NT, Visual Basic, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

PC Direct is a trademark of Ziff Communications Company in the United States, other countries, or both and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries or both and is licensed exclusively through X/Open Company Limited.

Other company, product, or service names, which may be denoted by a double asterisk (\*\*), may be trademarks or service marks of others.



---

# Index

## Special Characters

- #ifdefs, C/C++ language restrictions 510
- #include macro, C/C++ language restrictions 496
- #line macros, C/C++ language restrictions 496

## Numerics

- 64-bit integer (BIGINT) data type supported by DB2 Connect Version 6 682

## A

- access to data consideration
  - DB2 Call Level Interface (DB2 CLI) 25
  - embedded SQL 24
  - JDBC 25
  - Microsoft specifications 26
  - ODBC 25
  - REXX 24
  - using Perl 26
  - using query products 27
- ACQUIRE statement, DB2 Connect support 692
- activation time and triggers 378
- ActiveX Data Object specification supported in DB2 26
- ADHOC.SQC C program listing 162
- Administering Satellites Guide and Reference 724
- Administration Guide 721
- Administrative API Reference 721
- ADO specification
  - supported in DB2 26
- AFTER triggers 378, 383
- aggregating functions 259
- alerts, supported by triggers 374
- allocating dynamic memory in a UDF 340
- ALLOW PARALLEL clause 314
- ALTER NICKNAME statement
  - column options 475
  - data type mappings 479
- ambiguous cursors 684
- APPC, CPI-C and SNA Sense Codes 721
- APPC, handling interrupts 120
- Application Building Guide 721
- application design
  - access to data 23
  - binding 45
  - character conversion considerations 403
  - character conversion in SQL statements 403
  - character conversions in stored procedures 405
  - COBOL Japanese and traditional Chinese EUC considerations 590
  - COBOL requirements, include files 570
  - code points for special characters 404
  - coding a DB2
    - application, overview 9
  - collating sequences, guidelines 399
  - creating SQLDA structure, guidelines 152
  - cursor processing, considerations 80
  - data relationship 29
  - data value control consideration 27
  - declaring sufficient SQLVAR entities 148
  - describing SELECT statement 151
  - double-byte character support (DBCS) 404
  - dynamic SQL caching 60
  - error handling, guidelines 118
  - executing statements without variables 130
  - guidelines 22
  - input-SQLDA procedure, sample of 668
  - input-SQLDA stored procedure, sample of 675
  - logic at the server 31
  - OLE automation UDFs 317
  - package renaming 51
  - passing data, guidelines 156
  - precompiling and binding 45
  - receiving database values 73
- application design (*continued*)
  - retrieving data a second time 23
  - REXX requirements, registering routines 608
  - sample programs 104
  - saving end user requests 158
  - static SQL, advantages of using 60
  - table function
    - considerations 333
  - use of dynamic SQL, overview of 129
  - user-defined function (UDF) 287
  - using LOB locators in UDFs 335
  - using parameter markers 166
  - varying-list statements, processing of 158
- Application Development Guide 722
- application domain and object-orientation 221
- application environment, for programming 9
- application forms using CREATE TABLE example 277
- application logic
  - data relationship consideration 31
  - data value control consideration 29
- application logic consideration
  - stored procedures 31
  - triggers 32
  - user-defined functions 31
- Application Program Interface (API)
  - for JDBC applications 566
  - for setting contexts between threads
    - sqlAttachToCtx() 442
    - sqlBeginCtx() 442
    - sqlDetachFromCtx() 442
    - sqlEndCtx() 442
    - sqlGetCurrentCtx() 442
    - sqlInterruptCtx() 442
    - sqlSetTypeCtx() 442
  - overview of 35
  - restrictions in an XA environment 451
  - syntax for REXX 621

- Application Program Interface (API) *(continued)*
  - types of 566
  - uses of 35
- argument types, promotions in UDFs 303
- arguments, passing from DB2 to a UDF 288
- arguments between UDFs and DB2 289
  - call-type 294
  - dbinfo 297
  - diagnostic-message 293
  - function-name 292
  - scratchpad 293
  - specific-name 292
  - SQL-argument 289
  - SQL-argument-ind 290
  - SQL-result 289
  - SQL-result-ind 290
  - SQL-state 291
- ARI (DB2 for VSE & VM) 683
- arithmetic error
  - in UDFs 290
- AS LOCATOR clause 335
- ASCII
  - mixed-byte data 681
  - sort order 686
- assignments in dynamic SQL
  - example 283
- assignments involving different UDTs example 283
- assignments involving UDTs
  - example 282
- asynchronous events 441
- asynchronous nature of buffered insert 458
- ATOMIC compound SQL, not supported in DB2 Connect 690
- automation server, for OLE 318
- AVG over a UDT example 264

**B**

- backing out changes 19
- BASIC language
  - implementation of OLE automation UDF 318
- BASIC types and OLE automation types 320
- BEFORE triggers 378, 383
- BEGIN DECLARE SECTION 11
- beginning transactions 18
- big integer, data type 682
- BigDecimal Java type 535
- BIGINT parameter to UDF 304
- BIGINT SQL data type 75
- BIGINT SQL data type 525 *(continued)*
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- blob C/C++ type 525
- blob\_file C/C++ type 525
- BLOB-FILE COBOL type 586
- BLOB\_FILE FORTRAN type 603
- BLOB FORTRAN type 603
- blob\_locator C/C++ type 525
- BLOB-LOCATOR COBOL type 586
- BLOB\_LOCATOR FORTRAN type 603
- BLOB parameter to UDF 309
- BLOB SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- BLOBs (Binary Large Objects)
  - uses and definition 227
- blocking 684
- BSTR OLE automation type 320
- buffer size for buffered insert 455
- buffered insert
  - advantages 458
  - asynchronous nature of 458
  - buffer size 455
  - closed state 459
  - considerations for using 458
  - deadlock errors 459
  - error detection during 458
  - error reporting in buffered insert 459
  - group of rows 458
  - INSERT BUF bind option 458
  - long field restriction 460
  - not supported in CLP 460
  - open state 459
  - overview 455
  - partially filled 456
  - restrictions on using 460
  - savepoint consideration 456
  - SELECT buffered insert 459
  - statements that close 456
  - transaction log
    - consideration 458
  - unique key violation 459

**C**

- C++
  - consideration for stored procedures 208
  - considerations for UDFs 343
  - type decoration
    - consideration 492
- C++ types and OLE automation types 320
- C/C++ data types
  - blob 525
  - blob\_file 525
  - blob\_locator 525
  - char 525
  - clob 525
  - clob\_file 525
  - clob\_locator 525
  - dbclob 525
  - dbclob\_file 525
  - dbclob\_locator 525
  - double 525
  - float 525
  - long 525
  - long int 525
  - long long 525

- C/C++ data types *(continued)*
  - long long int 525
  - null-terminated character form 525
  - short 525
  - short int 525
  - sqldbchar 525
  - VARCHAR structured form 525
  - wchar\_t 525
- C/C++ language
  - character set 491
  - Chinese (Traditional) EUC considerations 524
  - data types supported 525
  - declaring graphic host variables 503
  - declaring host variables 499
  - embedding SQL statements 43
  - file reference declarations 509
  - handling class data members 517
  - handling null-terminated strings 514
  - host variable, naming 498
  - include files, required 493
  - initializing host variables 510
  - input and output files 492
  - Japanese EUC considerations 524
  - LOB data declarations 506
  - LOB locator declarations 508
  - member operator, restriction 518
  - pointer to data type, declaring in C/C++ 516
  - programming restrictions in 491
  - qualification operator, restriction 518
  - supported data types 525
  - trigraph sequences 491
- C language type definitions in `sqludf.h` 311
- C null-terminated graphic string SQL data type 320
- C null-terminated string SQL data type 320
- Call Level Interface (CLI)
  - advantages of using 178, 180
  - comparing embedded SQL and DB2 CLI 177
  - overview 177
- CALL statement
  - in Java 554
  - initializing client for stored procedure (DB2DARI) SQLDA structure 655
- CALL statement *(continued)*
  - invoking a stored procedure 554
  - on different platforms 689
- call-type 333
  - contents with scalar functions 295
  - contents with table functions 296
- call-type, passing to UDF 294
- CALL USING DESCRIPTOR statement (OS/400) 689
- calling convention
  - for UDFs 303
- calling from a REXX application 621
- calling the DB2 CLP from a REXX application 621
- CARDINALITY specification in table functions 333
- cascade 686
- cascading triggers 385
- CAST FROM clause 289
- CAST FROM clause in the CREATE FUNCTION statement 303
- castability 257
- casting, UDFs 273
- CHAR 291
- char C/C++ type 525
- CHAR parameter to UDF 305
- CHAR SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- CHAR type 566
- character comparison, overview 395
- character conversion
  - coding SQL statements 403
  - coding stored procedures 405, 432
  - during precompiling and binding 406
  - expansion 409
  - national language support (NLS) 407
  - programming considerations 403
  - rules for string conversions 432
  - string length overflow 431
- character conversion *(continued)*
  - string length overflow past data types 403
  - supported code pages 408
  - Unicode (UCS-2) 433
  - when executing an application 406
  - when occurs 407
- character host variables
  - C/C++
    - fixed and null-terminated 501
    - variable length 502
  - COBOL 577
  - FORTRAN 598
- Character Large Objects 227
- CHARACTER\*n FORTRAN type 603
- CHARACTER parameter to UDF 305
- character translation 681
- CHECKERR.CBL program listing 124
- CHECKERR REXX program listing 128
- Chinese (Traditional)
  - double-byte considerations 422
  - Extended UNIX Code considerations 422
- Chinese (Traditional) code sets 419
  - C/C++ considerations 524
  - developing applications using 422
- Chinese (Traditional) EUC code sets
  - REXX considerations 625
- CICS 680
- CICS SYNCPOINT ROLLBACK command 449
- classes
  - data members, as host variables in C/C++ 517
- CLASSPATH environment variable 558
- CLI 177
- CLI Guide and Reference 722
- client applications
  - running stored procedures 190
- client-based parameter validation
  - Extended UNIX Code consideration 428
- client/server
  - code page conversion 407
- clob C/C++ type 525
- clob\_file C/C++ type 525
- CLOB-FILE COBOL type 586

- CLOB\_FILE FORTRAN type 603
- CLOB FORTRAN type 603
- clob\_locator C/C++ type 525
- CLOB-LOCATOR COBOL type 586
- CLOB\_LOCATOR FORTRAN type 603
- CLOB parameter to UDF 309
- CLOB SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- CLOBs (Character Large Objects)
  - uses and definition 227
- CLOSE call 296
- closed state, buffered insert 459
- closing a buffered insert 456
- COBOL
  - declaring host variables 575
  - embedding SQL statements 43
  - file reference declarations 581
  - include files, list of 570
  - indicator tables 584
  - input and output files 569
  - Japanese and traditional Chinese EUC considerations 590
  - LOB data declarations 579
  - LOB locator declarations 581
  - object oriented restrictions 590
  - restrictions in 569
  - rules for indicator variables 579
  - supported data types 586
- COBOL data types
  - BINARY 585
  - BLOB 586
  - BLOB-FILE 586
  - BLOB-LOCATOR 586
  - CLOB 586
  - CLOB-FILE 586
  - CLOB-LOCATOR 586
  - COMP 585
  - COMP-1 586
  - COMP-3 586
  - COMP-4 585
  - COMP-5 586
  - DBCLOB 586
  - DBCLOB-FILE 586
  - DBCLOB-LOCATOR 586
  - PICTURE (PIC) clause 586
  - USAGE clause 586
- COBOL language
  - data types supported 586
- code page
  - allocating storage for unequal situations 425
  - binding considerations 53
  - character conversion 407
  - DB2CODEPAGE environment variable 401
  - handling expansion at application 428
  - handling expansion at server 428
  - how determined 401
  - in SQLERRMC field of SQLCA 683
  - locales
    - deriving in applications 402
    - how DB2 derives locales 402
  - national language support (NLS) 407
  - stored procedure considerations 208
  - supported Windows 95 code pages 401
  - supported Windows NT code pages 401
  - unequal situations 410, 425
- code point 394
- code point, definition of 394
- code set
  - in SQLERRMC field of SQLCA 683
- coding Java UDFs 313
- collating sequence
  - case independent comparisons 396
  - code point 394
  - collating\_sequence option 400
  - EBCDIC and ASCII 686
  - EBCDIC and ASCII sort order example 397
  - federated database concerns 400
  - general concerns 399
  - identity sequence 394
  - include files
    - in C/C++ 494
    - in COBOL 571
    - in FORTRAN 593
  - multi-byte characters 395
  - overview of 394
  - samples of 399
  - simulating EBCDIC binary collation 711
  - sort order example 397
- collating sequence (*continued*)
  - specifying 396
  - use in character comparisons 395
- collation
  - Chinese (Traditional) code sets 424
  - Japanese code sets 424
- collection ID 684
  - DB2 Universal Database for AS/400 685
- COLLECTION parameters 52
- collections 685
- column
  - setting null values in 74
  - supported SQL data types 75
  - using indicator variables on nullable data columns 78
- column functions 259
- column options
  - ALTER NICKNAME statement 475
  - description 474
- column types, creating in C/C++ 525
- column types, creating in COBOL 586
- column types, creating in FORTRAN 602
- COM.ibm.db2.app.Blob 665
- COM.ibm.db2.app.Clob 666
- COM.ibm.db2.app.Lob 665
- COM.ibm.db2.app.StoredProc 661
- COM.ibm.db2.app.UDF 313, 663
- COM.ibm.db2.jdbc.app.DB2Driver 540
- COM.ibm.db2.jdbc.net.DB2Driver 540
- Command Line Processor 621
- command line processor
  - prototyping utility 39
- Command Reference 722
- comments
  - SQL, rules for 573, 596
- comments, SQL, rules for 497
- COMMIT statement 11
  - association with cursor 80
  - ending a transaction 18
  - ending transactions 19
  - pass-through 485, 486
- COMMIT WORK RELEASE
  - not supported in DB2 Connect 693
- committing changes, tables 18
- COMP-1 in COBOL types 586
- COMP-3 in COBOL types 586
- COMP-5 in COBOL types 586



COMP and COMP-4 data types in  
     COBOL 585  
 comparisons involving UDTs  
     example 279, 281  
 compiled applications, creating  
     packages for 47  
 compiling 50  
     client application 210  
     stored procedures 209  
 compiling a UDF 260  
 completion code 15  
 compound SQL, NOT ATOMIC 690  
 configuration parameter  
     LOCKTIMEOUT 445  
 CONNECT  
     application programs 17  
 connect  
     CONNECT RESET  
         statement 682  
     CONNECT TO statement 682  
     implicit connect 682  
     null CONNECT 682  
 CONNECT  
     sample programs 104  
     SQLCA.SQLERRD settings 427  
 CONNECT RESET statement  
     ending transactions 19  
 CONNECT statement 11  
 CONNECT TYPE 2  
     considerations with stored  
         procedures 209  
 connecting DB2 application  
     programs 17  
 connection handle 177  
 Connectivity Supplement 722  
 consideration  
     access to data 23  
     application logic at server 31  
     data relationship control 29  
     data value control 27  
     DB2 application design 22  
 consistency  
     of data 17  
 consistency of data 17  
 consistent behavior and UDTs 275  
 constraint mechanisms on large  
     objects 221  
 contexts  
     application dependencies  
         between 444  
     database dependencies  
         between 444  
     preventing deadlocks  
         between 445  
     contexts (*continued*)  
         setting in multithreaded DB2  
             applications 444  
     control information to access large  
         object data 228  
     conventions used in this book 7  
 CONVERT  
     WCHARTYPE  
         in stored procedures 209  
 coordinator node  
     behavior without buffered  
         insert 457  
 cost of a UDT example 264  
 counter for UDFs example 353  
 counter OLE automation UDF object  
     example in BASIC 365  
 counter OLE automation UDF object  
     example in C++ 367  
 counting and defining UDFs  
     example 265  
 counting with an OLE automation  
     object 265  
 country code  
     in SQLERRMC field of  
         SQLCA 683  
 creatable multi-use OLE automation  
     server 323  
 creatable single-use OLE automation  
     server 323  
 CREATE DATABASE API  
     SQLEDBDESC structure 398  
 CREATE DISTINCT TYPE statement  
     and castability 257  
     examples of using 277  
     to define a UDT 276  
 CREATE FUNCTION MAPPING  
     statement  
         estimating overhead of invoking  
             functions 483  
         making data source functions  
             known to federated server 482  
         reducing overhead of invoking  
             functions 483  
         specifying function names 484  
 CREATE FUNCTION  
     statement 293, 294, 297, 333, 335  
     CAST FROM clause 303  
     for OLE automation UDFs 318  
     in federated systems 482  
     Java UDFs 314  
     RETURNS clause 303  
     to register a UDF 261  
 CREATE PROCEDURE  
     statement 190, 557  
 CREATE SERVER statement 328  
 CREATE STORGROUP statement  
     DB2 Connect support 681  
 CREATE TABLE statement  
     defining LOB columns 229  
     examples of using 277  
     lob-options-clause 229  
     tablespace-options-clause 229  
 CREATE TABLESPACE statement  
     DB2 Connect support 681  
 CREATE TRIGGER statement  
     multiple triggers 385  
     order of trigger activation 378  
     overview 375  
     REFERENCING clause 380  
 CREATE TYPE MAPPING  
     statement 478  
 CREATE USER MAPPING  
     statement 328  
 creating  
     Java stored procedures 557  
     Java UDFs 312  
     OLE automation UDFs 318  
 creating packages for compiled  
     applications 47  
 creator 684  
 critical section routine, in multiple  
     threads, guidelines 444  
 critical sections 444  
 ctr() UDF C program listing 353  
 CURRENT EXPLAIN MODE  
     register 52  
 CURRENT FUNCTION PATH  
     register 52  
 CURRENT QUERY OPTIMIZATION  
     register 52  
 cursor  
     ambiguous 91  
     completing a unit of work 80  
     declared WITH HOLD 80  
     declaring 80  
     FOR FETCH ONLY 90  
     issuing a COMMIT statement 80  
     naming, in REXX 610  
     naming and defining of 79  
     positioning at table end 104  
     processing, in dynamic SQL 134  
     processing, sample program 82,  
         136  
     processing, summary of 79  
     processing with SQLDA  
         structure 152  
     read-only 79, 90  
     read only, application  
         requirements for 80

- cursor (*continued*)
  - retrieving multiple rows
    - with 91, 79
  - sample program 91
  - updatable 91
  - use in CLI 177
- CURSOR.SQB COBOL program
  - listing 87
- CURSOR.SQC C program listing 83
- CURSOR.SQF FORTRAN program
  - listing 89
- Cursor.sqlj Java program listing 85
- cursor stability 687
- cursor usage in REXX 619
- cursors declared WITH HOLD
  - X/Open XA Interface 450

## D

- data
  - avoiding bottlenecks when
    - extracting 461
  - extracting large volumes 461
- data control language (DCL) 682
- data definition language (DDL) 681
- data manipulation language (DML) 681
- Data Movement Utilities Guide and Reference 722
- data relationship consideration
  - application logic 31
  - referential integrity 30
  - triggers 30
- data sources in federated systems
  - accessing tables, views 472
  - invoking functions 482
  - mapping data types from 477
  - mapping DB2 functions to 482
  - mapping isolation levels to 476
  - using distributed requests to query 479
  - using pass-through to query 485
- data structure
  - allocating for stored
    - procedures 190
  - manipulating for DB2DARI
    - stored procedure 657
  - SQLEDBDESC 398
  - user-defined, with multiple threads 443
- data structures, declaring 11
- data transfer
  - updating 104
- data type mappings
  - ALTER NICKNAME
    - statement 479

- data type mappings (*continued*)
  - CREATE TYPE MAPPING
    - statement 479
  - creating for data sources 478
  - creating for specific
    - columns 478
  - default 477
- data types
  - BLOBs 227
  - character conversion
    - overflow 432
  - class data members, declaring in
    - C/C++ 517
  - CLOB in C/C++ 530
  - CLOBs 227
  - conversion
    - between DB2 and
      - C/C++ 525
    - between DB2 and
      - COBOL 586
    - between DB2 and
      - FORTRAN 603
    - between DB2 and Java 535
    - between DB2 and OLE DB
      - table function 329
    - between DB2 and REXX 617
  - conversion between DB2 and
    - OLE automation 320
  - data value control
    - consideration 27
  - DBCLOBs 227
  - decimal
    - in FORTRAN 604
  - description 12
  - Extended UNIX Code
    - consideration 430
  - FOR BIT DATA
    - in C/C++ 530
  - FOR BIT DATA, in COBOL 589
  - how they are passed to a
    - UDF 303
  - in C/C++ 525
  - in COBOL 586
  - in FORTRAN 602
  - Java stored procedures
    - (DB2GENERAL) 659
  - list of types and their
    - representations in UDFs 303
  - numeric 681
  - object-oriented 221
  - pointer to, declaring in
    - C/C++ 516
  - selecting graphic types 520
  - SQL column types, list of 75

- data types (*continued*)
  - supported
    - in C/C++ 525, 530
    - in COBOL 586
    - in COBOL, rules for 588
    - in FORTRAN 602
    - in FORTRAN, rules for 604
    - in OLE automation
      - UDFs 320
    - VARCHAR in C/C++ 530
  - data value control consideration
    - application logic and variable
      - type 29
    - data types 27
    - referential integrity
      - constraints 28
    - table check constraints 28
    - views with check option 29
  - database access
    - using different contexts 441
    - using multiple threads 441
  - database creation, specifying
    - collating sequence 398
  - Database Descriptor Block (SQLEDBDESC), specifying
    - collating sequences 398
  - database manager APIs
    - calling using stored
      - procedures 187
    - defining, sample programs 104
  - DATE OLE automation type 320
  - DATE parameter to UDF 309
  - DATE SQL data type 75, 320
    - C/C++ 525
    - COBOL 586
    - FORTRAN 603
    - Java 535
    - Java stored procedures
      - (DB2GENERAL) 659
    - OLE DB table function 329
    - REXX 617
  - DB2 BIND command
    - creating packages 51
  - DB2 Call Level Interface (DB2 CLI)
    - access to data consideration 25
  - DB2 Connect 679
    - isolation level 688
  - DB2 Connect Enterprise Edition for
    - OS/2 and Windows NT Quick
      - Beginnings 725
  - DB2 Connect Enterprise Edition for
    - UNIX Quick Beginnings 725
  - DB2 Connect Personal Edition Quick
    - Beginnings 722

- DB2 Connect programming considerations 679
- DB2 Connect User's Guide 722
- DB2 Data Links Manager for AIX Quick Beginnings 725
- DB2 Data Links Manager for Windows NT Quick Beginnings 725
- DB2 Enterprise - Extended Edition for UNIX Quick Beginnings 724
- DB2 Enterprise - Extended Edition for Windows NT Quick Beginnings 725
- DB2 library
  - books 720
  - Information Center 728
  - language identifier for books 726
  - late-breaking information 727
  - online help 718
  - ordering printed books 732
  - printing PostScript books 730
  - searching online information 730
  - setting up document server 729
  - SmartGuides 717
  - structure of 717
  - viewing online information 727
- DB2 Personal Edition Quick Beginnings 724
- DB2 PREP command
  - example of 47
  - overview of 47
- DB2 program
  - set up 11
- DB2 Query Patroller Administration Guide 725
- DB2 Query Patroller Installation Guide 725
- DB2 Query Patroller User's Guide 726
- DB2 System Controller 210
- DB2 Universal Database for AS/400 FOR BIT DATA stored procedure considerations 208
- DB2 Universal Database for OS/390 FOR BIT DATA stored procedure considerations 208
- DB2 using DB2 Connect 679
- DB2Appl.java
  - application example 543
- DB2ARXCS.BND REXX bind file 620
- DB2ARXNC.BND REXX bind file 620
- DB2ARXRR.BND REXX bind file 620
- DB2ARXRS.BND REXX bind file 620
- DB2ARXUR.BND REXX bind file 620
- db2bfd, bind file dump utility 54
- db2dari executable 198
- DB2DARI stored procedures 193
- db2diag.log file 468
- DB2GENERAL stored procedures 193
- DB2INCLUDE environment variable 496, 574, 595
- db2nodes.cfg file 469
- db2udf executable 342
- DB2Udf.java 312
- DBCLOB
  - Chinese (Traditional) code sets 424
  - Japanese code sets 424
- dbclob C/C++ type 525
- dbclob\_file C/C++ type 525
- DBCLOB-FILE COBOL type 586
- dbclob\_locator C/C++ type 525
- DBCLOB-LOCATOR COBOL type 586
- DBCLOB parameter to UDF 310
- DBCLOB SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- DBCLOB type 566
- DBCLOBs (Double-Byte Character Large Objects)
  - uses and definition 227
- DBCS 419
- dbinfo 333
- dbinfo, passing to UDF 297
- DBINFO keyword 297
- dbminfo argument, elements of 297
  - (ver\_rel) version/release number 299
  - appl\_id (unique application identifier) 300
  - application authorization ID (authid) 298
  - application authorization ID length (authidlen) 298
  - authid (application authorization ID) 298
- dbminfo argument, elements of 298
  - (continued)
  - authidlen (application authorization ID length) 299
  - codepg (database code page) 298
  - colname (column name) 298
  - colnamelen (column name length) 298
  - column name (colname) 298
  - column name length (colnamelen) 298
  - data base name (dbname) 297
  - data base name length (dbnamelen) 297
  - database code page (codepg) 298
  - dbname (data base name) 297
  - dbnamelen (data base name length) 297
  - numtfcol (table function columns entries) 299
  - platform 299
  - schema name (tbschema) 298
  - schema name length (tbschemalen) 298
  - table function column list (tfcolumn) 300
  - table function columns entries (numtfcol) 299
  - table name (tname) 298
  - table name length (tbnamelen) 298
  - tname (table name) 298
  - tbnamelen (table name length) 298
  - tbschema (schema name) 298
  - tbschemalen (schema name length) 298
  - tfcolumn (table function column list) 300
  - unique application identifier (appl\_id) 300
  - version/release number (ver\_rel) 299
- DCL (data control language) 682
- DDL (data definition language) 681
- deadlocks
  - error in buffered insert 459
  - in multithreaded applications 444
  - preventing in multiple contexts 445
- debugging
  - Java programs 538

- debugging (*continued*)
  - SQLJ programs 538
  - stored procedures 210, 214
    - using Stored Procedure Builder 559
  - using Visual Studio 214
- debugging your UDF 371
- DECIMAL parameter to UDF 304
- DECIMAL SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- DECLARE CURSOR statement 16
- DECLARE CURSOR statement, overview of 79
- DECLARE PROCEDURE statement (OS/400) 689
- declare section
  - in C/C++ 499, 528
  - in COBOL 575, 588
  - in FORTRAN 597, 604
  - rules for statements 69
- DECLARE statement
  - DB2 Connect support 692
- DECLARE STATEMENT
  - DB2 Connect support 693
- declaring
  - host variable, rules for 69
  - indicator variables 73
- deferring the evaluation of a LOB expression example 239
- defining the UDT and UDFs example 223
- DELETE
  - DB2 Connect support 681
  - triggers 380
- DELETE operation and triggers 376
- DESCRIBE statement
  - DB2 Connect support 693
  - double-byte character set consideration 429
  - Extended UNIX Code consideration 429
  - Extended UNIX Code consideration with EUC database 430
  - processing arbitrary statements 157
- descriptor handle 177
- designing DB2 applications, guidelines 22
- DFT\_SQLMATHWARN
  - configuration parameter 290
- diagnostic-message, passing to UDF 293
- differences between different DB2 products 680
- differences between host or AS/400 server and workstation 692, 693
- differences in SQLCODEs and SQLSTATES 687
- directed DSS 454
- distinct type 257
- distributed environment 679
- distributed requests
  - coding 479
  - optimizing 481
- divid() UDF C program listing 344
- DML (data manipulation language) 681
- Double-Byte Character Large Objects 227
- double-byte character set
  - Chinese (Traditional) code sets 422
  - configuration parameters 418
  - considerations for collation 424
  - Japanese code sets 422
  - mixed code set environments 424
  - Traditional Chinese considerations 422
  - unequal code pages 425
- double-byte character set (DBCS)
  - Chinese (Traditional) code sets 419
  - Japanese code sets 419
- double-byte code pages 421
- double C/C++ type 525
- double Java type 535
- double OLE automation type 320
- DOUBLE parameter to UDF 304, 305
- DOUBLE PRECISION parameter to UDF 305
- DOUBLE SQL data type 75, 320
- DSN (DB2 Universal Database for OS/390) 683
- DUOW 434
- DYNAMIC.CMD REXX program listing 146
- dynamic cursors 684
- Dynamic.java Java program listing 140
- dynamic memory, allocating in the UDF 340
- DYNAMIC.SQB COBOL program listing 142
- DYNAMIC.SQC C program listing 138
- DYNAMIC.SQF FORTRAN program listing 144
- dynamic SQL
  - arbitrary statements, processing of 157
  - authorization considerations 33
  - comparing to static SQL 130
  - considerations 130
  - contrast with dynamic SQL 60
  - cursor processing 134
  - cursor processing, sample program 136
  - DB2 Connect support 679
  - declaring an SQLDA 148
  - EXECUTE IMMEDIATE statement, summary of 130
  - EXECUTE statement, summary of 130
  - limitations 130
  - overview 129
  - parameter markers in 166
  - PREPARE statement, summary of 130
  - supported statements, list of 129
  - syntax, differences with static 130
  - using PREPARE, DESCRIBE, and FETCH 133
- dynamic statements
  - binding 52

## E

- easier maintenance using triggers 374
- EBCDIC
  - mixed-byte data 681
  - sort order 686
- embedded SQL
  - access to data consideration 24
- embedded SQL statement
  - comments, rules for 497
  - examples of 44
  - overview of 43
  - rules for, in C/C++ 497
  - rules for, in COBOL 573
  - rules for, in FORTRAN 596
  - syntax rules 44
- embedded SQL statements
  - comments, rules for 573, 596

- embedded SQL statements
  - (continued)
  - host variable, referencing in 573
- encapsulation and UDTs 275
- END DECLARE SECTION 11
- ending transactions 18
- ending transactions implicitly 20
- environment APIs
  - include file for C/C++ 494
  - include file for COBOL 571
  - include file for FORTRAN 593
- environment,for programming 9
- environment handle 177
- error code 15
- error detection in a buffered
  - insert 458
- error handling
  - C/C++ language
    - precompiler 496
  - considerations in a partitioned
    - environment 467
  - during precompilation 49
  - identifying partition that returns
    - the error 469
  - in a looping application 469
  - in a suspended application 469
  - include file
    - for C/C++ applications 495
    - for FORTRAN 593, 595
    - in COBOL 570, 572
  - merged multiple SQLCA
    - structures 468
  - overview of 117
  - reporting 468
  - resetting 16
  - SQLCA structure 468
  - SQLCODE 468
  - using WHENEVER
    - statement 118
  - WHENEVER statement 15
  - with the SQLCA 14
- error messages
  - error conditions flag 117
  - exception condition flag 117
  - SQLCA structure 117
  - SQLSTATE 117
  - SQLWARN structure 117
  - timestamps, when
    - precompiling 56
  - warning condition flag 117
- EUC 419
- examples
  - ADHOC.SQC C program
    - listing 162
- examples (continued)
  - application forms using CREATE
    - TABLE 162
  - assignments in dynamic
    - SQL 283
  - assignments involving different
    - UDTs 283
  - assignments involving
    - UDTs 282
  - AVG over a UDT 264
  - BLOB data declarations 507
  - CHECKERR.CBL program
    - listing 124
  - CHECKERR REXX program
    - listing 128
  - CLOB data declarations 507
  - CLOB file reference 509
  - CLOB locator 509
  - comparisons involving
    - UDTs 279, 281
  - cost of a UDT 264
  - counter for UDFs 353
  - counter OLE automation UDF
    - object in BASIC 365
  - counter OLE automation UDF
    - object in C++ 367
  - counting and defining
    - UDFs 265
  - counting with an OLE
    - automation object 265
  - ctr() UDF C program listing 353
  - CURSOR.SQB COBOL program
    - listing 87
  - CURSOR.SQC C program
    - listing 83
  - CURSOR.SQF FORTRAN
    - program listing 89
  - Cursor.sqlj Java program
    - listing 85
  - DB2Appl.java 543
  - DBCLOB data declarations 507
  - declaring BLOB file references
    - using COBOL 581
  - declaring BLOB file references
    - using FORTRAN 602
  - declaring BLOB locator using
    - COBOL 581
  - declaring BLOBs using
    - COBOL 580
  - declaring BLOBs using
    - FORTRAN 600
  - declaring CLOB file locator using
    - FORTRAN 601
  - declaring CLOBs using
    - COBOL 580
- examples (continued)
  - declaring CLOBs using
    - FORTRAN 162
  - declaring DBCLOBs using
    - COBOL 580
  - deferring the evaluation of a LOB
    - expression 239
  - defining the UDT and UDFs 223
  - divid() UDF C program
    - listing 344
  - DYNAMIC.CMD REXX program
    - listing 146
  - Dynamic.java Java program
    - listing 140
  - DYNAMIC.SQB COBOL program
    - listing 142
  - DYNAMIC.SQC C program
    - listing 138
  - DYNAMIC.SQF FORTRAN
    - program listing 144
  - exploiting LOB function to
    - populate the database 225
  - exploiting UDFs to query
    - instances of UDTs 225
  - exponentiation and defining
    - UDFs 261
  - extracting a document to a file
    - (CLOB elements in a
      - table) 248
  - extracting large volumes of data
    - program listing 461
  - find the vowel, fold the
    - CLOB 348
  - findvwl() UDF C program
    - listing 348
  - fold() UDF C program
    - listing 348
  - function invocations 268
  - function using LOB locator 362
  - implementation of OLE
    - automation UDF
      - in BASIC language 318, 321
      - in C++ language 321, 323
  - INPCLISQF FORTRAN program
    - listing 671
  - INPSRV.CMD REXX program
    - listing 676
  - inserting data into a CLOB
    - column 253
  - integer divide operator for
    - UDFs 344
  - Java applets 543
  - LOBEVAL.SQB COBOL program
    - listing 243



examples (*continued*)

- LOBEVAL.SQC C program
  - listing 162
- LOBFILE.SQB COBOL program
  - listing 251
- LOBFILE.SQC C program
  - listing 249
- LOBLOC.SQB COBOL program
  - listing 234
- LOBLOC.SQC C program
  - listing 232
- LOBLOC.SQF FORTRAN
  - program listing 237
- mail OLE automation UDF object
  - in BASIC 369
- money using CREATE DISTINCT
  - TYPE 277
- OPENFTCH.SQB COBOL
  - program listing 97
- OPENFTCH.SQC C program
  - listing 93
- OPENFTCH.SQF FORTRAN
  - program listing 99
- Openftch.sqlj Java program
  - listing 95
- Outcli.java Java program
  - listing 204
- Outsrv.sqlj Java program
  - listing 207
- parameter markers in
  - functions 269
- registering SQLEXEC, SQLDBS
  - and SQLDB2
    - on AIX 609
  - registering SQLEXEC, SQLDBS
    - and SQLDB2 for REXX
      - on OS/2 608
      - on Windows 608
- resume using CREATE
  - DISTINCT TYPE 277
- sales using CREATE TABLE 277
- sample SQL declare section for
  - supported SQL data types 528
- search string and BLOBs 263
- SQLException handling 123
- STATIC.SQB COBOL program
  - listing 66
- STATIC.SQC C program
  - listing 64
- STATIC.SQF FORTRAN program
  - listing 67
- Static.sqlj Java program
  - listing 65
- static update 104

examples (*continued*)

- string search and defining
  - UDFs 162
- string search over UDT 263
- syntax for character host
  - variables in FORTRAN 598, 599
- table function returning
  - document IDs 266
- tfweather\_u table function C
  - program listing 355
- unqualified function
  - reference 270
- UPDAT.CMD REXX program
  - listing 115
- UPDAT.SQB COBOL program
  - listing 111
- UPDAT.SQC C program
  - listing 107
- UPDAT.SQF FORTRAN program
  - listing 113
- Updat.sqlj Java program
  - listing 109
- use of UDTs in UNION 284
- user-defined sourced functions on
  - UDTs 282
- using a locator to work with a
  - CLOB value 231
- using class data members in an
  - SQL statement 517
- using GET ERROR
  - MESSAGE 120
- using indicator variables 78
- using parameter markers in
  - search and update 167
- using qualified function
  - reference 269
- UTIL.C program listing 120
- UTIL.F program listing 126
- Varinp.java Java program
  - listing 171
- VARINP.SQB COBOL program
  - listing 173
- VARINP.SQC C program
  - listing 169
- VARINP.SQF FORTRAN program
  - listing 175
- EXCSQLSTT, host or AS/400
  - command 693
- EXEC SQL INCLUDE SQLCA
  - multithreading
    - considerations 443
- EXEC SQL INCLUDE statement,
  - C/C++ language restrictions 496

- EXECUTE IMMEDIATE statement,
  - summary of 130
- EXECUTE statement, summary
  - of 130
- execution requirements for
  - REXX 619
- exit routines, use restrictions 120
- expansion of data on the host or
  - AS/400 server 681
- EXPLAIN, prototyping utility 39
- Explain Snapshot 53
- exploiting
  - LOB function to populate the
    - database example 225
  - UDFs to query instances of UDTs
    - example 225
- EXPLSNAP bind option 53
- exponentiation and defining UDFs
  - example 261
- extended dynamic SQL statements
  - not supported in DB2
    - Connect 693
- Extended UNIX Code (EUC)
  - character conversion
    - overflow 431
  - character conversions in stored
    - procedures 432
  - character string length
    - overflow 432
  - Chinese (Traditional)
    - C/C++ consideration 524
    - REXX consideration 625
  - Chinese (Traditional) code
    - sets 419, 422
  - client-based parameter
    - validation 428
  - considerations for collation 424
  - considerations for DBCLOB
    - files 424
  - double-byte code pages 421
  - expansion at application 428
  - expansion at server 428
  - expansion samples 428
  - fixed or varying length data
    - types 430
  - graphic constants 423
  - graphic data handling 423
  - Japanese
    - C/C++ consideration 524
    - REXX consideration 625
  - Japanese and traditional Chinese
    - COBOL consideration 590
    - FORTRAN consideration 605
  - Japanese code sets 419, 422
  - mixed code pages 421

- Extended UNIX Code (EUC)
  - (continued)
  - mixed code set
    - environments 431
  - rules for string conversions 432
  - stored procedures
    - considerations 424
  - Traditional Chinese
    - considerations 422
  - UDF considerations 423
  - unequal code pages 425
  - using the DESCRIBE statement 429
- extensibility and UDTs 275
- extern declaration
  - C++ 492
- EXTERNAL ACTION option and UDFs 340
- EXTERNAL clause 191
- EXTERNAL NAME clause 327, 328
- extracting
  - large volumes of data 461
- extracting a document to a file (CLOB elements in a table)
  - example 248
- F**
- faster application development using triggers 374
- federated database
  - collating sequences, guidelines 400
- federated systems
  - column options 474
  - data integrity 476
  - data source functions 482
  - data source tables, views
    - cataloging information about 472
    - considerations, restrictions 473
    - nicknames for 472
  - data type mappings 477
  - distributed requests 479
  - function mapping options 483
  - function mappings 482
  - introduction 471
  - isolation levels 476
  - nicknames 472
  - pass-through 485
  - server options 481
- FENCED option and UDFs 340
- FETCH call 296
- FETCH statement
  - host variables 133
- FETCH statement (continued)
  - repeated access, technique for 133
  - scroll backwards, technique for 101
  - using SQLDA structure with 151
- file extensions
  - sample programs 633
- file reference declarations in REXX 615
- file reference variables
  - examples of using 248
  - for manipulating LOBs 227
  - input values 246
  - output values 247
- final call, to a UDF 294
- FINAL CALL clause 296
- FINAL CALL keyword 294
- finalize Java method 314
- find the vowel, fold the CLOB for UDFs example 348
- findvwl() UDF C program
  - listing 348
- FIPS 127-2 standard 15
- FIRST call 296
- first call, to a UDF 294
- fixed or varying length data types
  - Extended UNIX Code consideration 430
- flagger utility, used in precompiling 50
- flexibility and UDTs 275
- float C/C++ type 525
- float OLE automation type 320
- FLOAT parameter to UDF 305
- FLOAT SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- floating point parameter to UDF 305
- flushed buffered insert 456
- fold() UDF C program listing 348
- FOR BIT DATA
  - considerations for stored procedures 208
  - data type in C/C++ 530
- FOR BIT DATA modifier in UDFs 306
- FOR BIT DATA SQL data type 320
- FOR EACH ROW trigger 377
- FOR EACH STATEMENT trigger 377
- FOR UPDATE clause 90
- FORCE command 683
- foreign key, DB2 Connect support 686
- FORTRAN
  - data types supported 602, 604
  - embedding SQL statements 43
  - file reference declarations 601
  - host variables, overview 597
  - indicator variables, rules for 600
  - input and output files 592
  - Japanese and traditional Chinese
    - EUC considerations 605
  - LOB data declarations 600
  - LOB locator declarations 601
  - programming
    - considerations 591
    - programming restrictions 591
    - referencing host variables 596
- FORTRAN data types
  - BLOB 603
  - BLOB\_FILE 603
  - BLOB\_LOCATOR 603
  - CHARACTER\*n 603
  - CLOB 603
  - CLOB\_FILE 603
  - CLOB\_LOCATOR 603
  - INTEGER\*2 603
  - INTEGER\*4 603
  - REAL\*2 603
  - REAL\*4 603
  - REAL\*8 603
- FORTRAN language
  - data types supported 602
- fullselect
  - buffered insert
    - consideration 460
- fullselect consideration 460
- fully qualified name 327
- FUNCPATH bind option 53
- function directory 191
- function invocations example 268
- function mappings
  - CREATE FUNCTION MAPPING statement 482, 483
  - options 483
- function-name, passing to UDF 292
- function path and UDFs 258
- function references, summary for UDFs 271

- function selection algorithm and UDFs 258
- function templates 482
- functions
  - aggregating functions 259
  - column functions 259
  - scalar functions 259
  - syntax for referring to 267
  - table functions 259
  - with SQL triggered statements 383

## G

- GENERAL stored procedures 193
- GENERAL WITH NULLS stored procedures 193
- GET ERROR MESSAGE API 612
- getAsciiStream JDBC API 566
- getString JDBC API 566
- getUnicodeStream JDBC API 566
- global enforcement of business rules using triggers 374
- Glossary 723
- GRANT statement
  - DB2 Connect support 682
- graphic constants
  - Chinese (Traditional) code sets 423
  - Japanese code sets 423
- graphic data
  - Chinese (Traditional) code sets 419, 423
  - Japanese code sets 419, 423
- graphic data types
  - selecting 520
- graphic host variables
  - C/C++ 503
  - COBOL 578
- GRAPHIC parameter to UDF 307
- GRAPHIC SQL data type
  - C/C++ 525
  - COBOL 586
  - FORTRAN, not supported in 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- graphic strings
  - character conversion 410
- GRAPHIC type 566
- graphical objects
  - considerations for Java 566

- GROUP BY clause
  - sort order 686
- group of rows
  - in buffered insert 458
- guideline
  - access to data 23
  - application logic at server 31
  - data relationship control 29
  - data value control 27
  - DB2 application design 22

## H

- handle
  - connection handle 177
  - descriptor handle 177
  - environment handle 177
  - statement handle 177
- host or AS/400
  - accessing host servers 441
- Host or AS/400 environment, programming 679
- host variables
  - allocating in stored procedures 189
  - class data members, handling in C/C++ 517
  - clearing LOB host variables in REXX 617
  - considerations for stored procedures 209
  - declaring 69
    - in C/C++ 499
    - in COBOL 575
    - in FORTRAN 597
  - declaring, examples of 72
  - declaring, rules for 69
  - declaring, sample programs 104
  - declaring as pointer to data type 516
  - declaring graphic
    - in C/C++ 503
    - in COBOL 578
  - declaring using variable list statement 158
  - definition 69
  - determining how to define for use with a column 14
  - file reference declarations in C/C++ 509
  - file reference declarations in COBOL 581
  - file reference declarations in FORTRAN 601
  - file reference declarations in REXX 615

- host variables (*continued*)
  - FORTRAN, overview of 189
  - graphic data 519
  - handling graphic
    - multi-byte character encoding 519
    - WCHARTYPE precompiler option 521
  - in REXX 611
  - initializing for stored procedure 189
  - initializing in C/C++ 510
  - LOB data declarations in C/C++ 506
  - LOB data declarations in COBOL 579
  - LOB data declarations in FORTRAN 600
  - LOB data in REXX 614
  - LOB locator declarations in C/C++ 508
  - LOB locator declarations in COBOL 581
  - LOB locator declarations in FORTRAN 601
  - LOB locator declarations in REXX 615
  - naming
    - in C/C++ 498
    - in COBOL 575
    - in FORTRAN 597
    - in REXX 611
  - null-terminated strings, handling in C/C++ 514
  - referencing
    - in C/C++ 498
    - in COBOL 575
    - in FORTRAN 596
    - in REXX 611
  - referencing from SQL, examples 72
  - relating to an SQL statement 13
  - selecting graphic data types 520
  - static SQL 69
  - use in dynamic SQL 129
  - used to pass blocks of data 452
- how to use this book 4
- HTML page
  - tagging for Java applets 543
- I
- identity sequence 394
- implementing a UDF 260
- implicit connect 682



- IN stored procedure parameters 191, 196
- include file
  - C/C++ requirements for 493
  - COBOL requirements for 570
  - FORTRAN requirements for 592
- SQL
  - C/C++ 493
  - COBOL 570
  - FORTRAN 592
- SQL1252A
  - COBOL 572
  - FORTRAN 594
- SQL1252B
  - COBOL 572
  - FORTRAN 594
- SQLADEF
  - C/C++ 493
- SQLAPREP
  - C/C++ 493
  - COBOL 570
  - FORTRAN 592
- SQLCA
  - C/C++ 493
  - COBOL 570
  - FORTRAN 592
- SQLCA\_92
  - COBOL 570
  - FORTRAN 593
- SQLCA\_CN
  - FORTRAN 592
- SQLCA\_CS
  - FORTRAN 592
- SQLCLI
  - C/C++ 493
- SQLCLI1
  - C/C++ 493
- SQLCODES
  - C/C++ 493
  - COBOL 570
  - FORTRAN 593
- SQLDA
  - C/C++ 494
  - COBOL 571
  - FORTRAN 593
- SQLDACT
  - FORTRAN 593
- SQLE819A
  - C/C++ 494
  - COBOL 571
  - FORTRAN 593
- SQLE819B
  - C/C++ 494
  - COBOL 571
  - FORTRAN 594

- include file (*continued*)
  - SQLE850A
    - C/C++ 494
    - COBOL 571
    - FORTRAN 594
  - SQLE850B
    - C/C++ 494
    - COBOL 571
    - FORTRAN 594
  - SQLE932A
    - C/C++ 494
    - COBOL 572
    - FORTRAN 594
  - SQLE932B
    - C/C++ 495
    - COBOL 572
    - FORTRAN 594
  - SQLEAU
    - C/C++ 494
    - COBOL 571
    - FORTRAN 593
  - SQLENV
    - C/C++ 494
    - COBOL 571
    - FORTRAN 593
  - SQLETSB
    - COBOL 571
  - SQLEXT
    - C/C++ 494
  - SQLJACB
    - C/C++ 495
  - SQLMON
    - C/C++ 495
    - COBOL 572
    - FORTRAN 594
  - SQLMONCT
    - COBOL 572
  - SQLSTATE
    - C/C++ 495
    - COBOL 572
    - FORTRAN 595
  - SQLSYSTM
    - C/C++ 495
  - SQLUDF
    - C/C++ 495
  - SQLUTBCQ
    - COBOL 572
  - SQLUTBSQ
    - COBOL 572
  - SQLUTIL
    - C/C++ 495
    - COBOL 573
    - FORTRAN 595
  - SQLUV
    - C/C++ 495

- include file (*continued*)
  - SQLUVEND
    - C/C++ 495
  - SQLXA
    - C/C++ 495
- include files
  - locating in C/C++ 496
  - locating in COBOL 574
  - locating in FORTRAN 595
- INCLUDE SQLCA
  - pseudocode 14
- INCLUDE SQLDA statement 16
- INCLUDE SQLDA statement, creating SQLDA structure 152
- INCLUDE statement 16
- inconsistent data 17
- inconsistent state 17
- indicator tables, COBOL support for 584
- indicator variables
  - declaring 73
  - in C/C++, rules for 503
  - in COBOL, rules for 579
  - in FORTRAN, rules for 600
  - in REXX, rules for 612
  - using in DB2DARI stored procedures 657
  - using on nullable columns 78
- indicator variables and LOB locators 246
- infix notation and UDFs 271
- INOUT stored procedure parameters 191
- INOUT stored procedure parameter 197
- INPCLISQF FORTRAN program listing 671
- INPSRV.CMD REXX program listing 676
- input and output files
  - C/C++ 492
  - COBOL 569
  - FORTRAN 592
- input and output to screen and keyboard and UDFs 342
- input file extensions, C/C++ language 492
- INSERT BUF bind option
  - buffered inserts 458
- INSERT operation and triggers 376
- INSERT statement
  - DB2 Connect support 681, 682
  - not supported in CLP 460
  - with VALUES clause 457

- inserting data into a CLOB column
  - example 253
- inserts
  - without buffered insert 457
- Installation and Configuration Supplement 723
- instances of object-oriented data types, storing 221
- Int Java type 535
- INTEGER 294
- INTEGER\*2 FORTRAN type 603
- INTEGER\*4 FORTRAN type 603
- integer divide operator for UDFs
  - example 344
- INTEGER or INT parameter to UDF 304
- INTEGER SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- interactive SQL
  - processing, sample program 159
- interrupt handling with SQL statements 119
- interrupts, SIGUSR1 120
- invoking UDFs 267
- ISO 10646 standard 419
- ISO 2022 standard 419
- ISO/ANS SQL92 686
- ISO/ANS SQL92 standard 15
- isolation level 476, 688

**J**

- Japanese and traditional Chinese EUC code sets
  - COBOL considerations 590
  - FORTRAN considerations 605
- Japanese code sets 419
  - C/C++ considerations 524
  - developing applications using 422
- Japanese EUC code sets
  - REXX considerations 625
- Java
  - applet support 539
  - application support 539
  - comparison of SQLJ with JDBC 533
  - comparison with other languages 534
  - debugging 538

- Java (*continued*)
  - distributing and running
    - applets 539
  - distributing and running applications 543
  - embedding SQL statements 43
  - installing JAR files 562, 563
  - JDBC
    - example program using 541
  - JDBC specification 538
  - overview 533
  - overview of DB2 support for 538
  - SQLJ (Embedded SQL for Java)
    - calling stored procedures 554
    - example program using 550
    - host variables 554
  - SQLJ (Embedded SQLJ for Java) 545
    - applets 547
    - db2prof 546
    - db2profp 546
    - declaring cursors 549
    - declaring iterators 549
    - embedding SQL statements in 548
    - example clauses 548
    - positioned DELETE statement 549
    - positioned UPDATE statement 549
    - profconv 546
    - restrictions 547
    - translator 546
  - SQLJ specification 538
  - stored procedures 562, 563
    - examples 564
  - UDFs (user-defined functions) 562, 563
    - examples 564

- Java application
  - SCRATCHPAD
    - consideration 314
  - signature for UDFs 313
  - using graphical and large objects 566

- Java class files
  - CLASSPATH environment variable 558
  - java\_heap\_sz configuration parameter 558
  - jdk11\_path configuration parameter 558
  - where to place 558

- Java data types
  - BigDecimal 535
  - Blob 535
  - double 535
  - Int 535
  - java.math.BigDecimal 535
  - short 535
  - String 535
- Java Database Connectivity 540
- java\_heap\_sz configuration parameter 558
- Java I/O streams
  - System.err 313
  - System.in 313
  - System.out 313
- java.math.BigDecimal Java type 535
- Java packages and classes 540
  - COM.ibm.db2.app 535
- JAVA stored procedures 193
- Java UDF consideration 288
- JDBC
  - access to data consideration 25
  - APIs
    - getAsciiStream 566
    - getString 566
    - getUnicodeStream 566
    - setAsciiStream 566
    - setString 566
    - setUnicodeStream 566
  - comparison with SQLJ 533
  - drivers
    - COM.ibm.db2.jdbc.app.DB2Driver 540
    - COM.ibm.db2.jdbc.net.DB2Driver 540
  - installing 545
  - version 1.22 544
  - version 2.0 544
  - example program using 541
  - SQLJ interoperability 567
- jdk11\_path configuration parameter 558

**L**

- LABEL ON statement
  - DB2 Connect support 692
- LANGUAGELEVEL precompile option MIA
  - SAA1 530
  - using SQL92E and SQLSTATE or SQLCODE variables 530, 589, 604
- LANGUAGELEVEL SQL92E precompile option 686
- LANGUAGE OLE clause 318
- large object data type 682
- large object descriptor 227
- large object value 227

- latch
  - status with multiple threads 442
- limitations
  - stored procedures (DB2DARI) 657
- linking
  - client application 210
  - overview of 50
  - stored procedures 209
- linking a UDF 260
- LOB data type
  - supported by DB2 Connect Version 6 682
- LOB locator APIs, used in UDFs
  - sqludf\_append API 335
  - sqludf\_create\_locator API 335
  - sqludf\_free\_locator API 335
  - sqludf\_length API 335
  - sqludf\_substr API 335
- LOB locator example program
  - listing 362
- LOB locators
  - scenarios for using 339
  - used in UDFs 335
- lob-options-clause of the CREATE TABLE statement 229
- LOBEVAL.SQB COBOL program
  - listing 243, 251
- LOBEVAL.SQC C program
  - listing 241, 249
- LOBLOC.SQB COBOL program
  - listing 234
- LOBLOC.SQC C program
  - listing 232
- LOBLOC.SQF FORTRAN program
  - listing 237
- LOBs (Large Objects)
  - and DB2 object extensions 221
  - considerations for Java 566
  - file reference variables 227
    - examples of using 248
    - input values 246
    - output values 247
  - SQL\_FILE\_APPEND, output value option 247
  - SQL\_FILE\_CREATE, output value option 247
  - SQL\_FILE\_OVERWRITE, output value option 247
  - SQL\_FILE\_READ, input value option 246
  - large object descriptor 227
  - large object value 227
  - locators 227, 229
    - example of using 231, 239
  - LOBs (Large Objects) *(continued)*
    - locators 221, 229 *(continued)*
      - indicator variables 231
      - programming scenarios 239
    - manipulating 221
    - programming options for values 230
    - storing 221
    - synergy with triggers, UDTs, and UDFs 386
    - synergy with UDTs and UDFs
      - examples of complex applications 223
  - local bypass 454
  - locales
    - deriving in application programs 402
    - how DB2 derives 402
  - locating include files
    - C/C++ 496
    - COBOL 574
    - FORTRAN 595
  - locators for manipulating LOBs 227
  - locking 687
    - buffered insert error 459
  - LOCKTIMEOUT configuration
    - parameter 445
  - LOCKTIMEOUT multisite update
    - configuration parameter 439
  - long C/C++ type 525
  - long field restriction
    - using buffered inserts 460
  - long fields 682
  - long int C/C++ type 525
  - long long C/C++ type 525
  - long long int C/C++ type 525
  - long OLE automation type 320
  - LONG VARCHAR
    - parameter to UDF 306
    - storage limits 227
  - LONG VARCHAR SQL data
    - type 75, 320
      - C/C++ 525
      - COBOL 586
      - FORTRAN 603
      - Java 535
      - Java stored procedures (DB2GENERAL) 659
      - OLE DB table function 329
      - REXX 617
  - LONG VARGRAPHIC
    - parameter to UDF 308
    - storage limits 227
  - LONG VARGRAPHIC SQL data
    - type 75, 320
- LONG VARGRAPHIC SQL data
  - type 525, 320 *(continued)*
    - C/C++ 525
    - COBOL 586
    - FORTRAN 603
    - Java 535
    - Java stored procedures (DB2GENERAL) 659
    - OLE DB table function 329
    - REXX 617
- LONGVAR type 566
- looping application
  - diagnosing 469

## M

- macro processing for the C/C++ language 491
- macros in sqludf.h 311
- mail OLE automation UDF object
  - example in BASIC 369
- manipulating large objects 221
- maxdari configuration
  - parameter 557
- maximum size for large object
  - columns, defining 228
- member operator, C/C++
  - restriction 518
- memory
  - decreasing requirement using LOB locators 335
- memory, allocating dynamic in the UDF 340
- memory allocation for unequal code pages 425
- memory size, shared for UDFs 342
- message file, definition of 49
- Message Reference 723
- method invocation
  - for OLE automation UDFs 319
- MIA 530
- Microsoft Exchange, used in mail
  - example 369
- Microsoft specifications
  - access to data consideration 26
  - ADO (ActiveX Data Object) 26
  - MTS (Microsoft Transaction Server) 26
  - RDO (Remote Data Object) 26
  - Visual Basic 26
  - Visual C++ 26
- Microsoft Transaction Server
  - specification
    - access to data consideration 26
- mixed-byte data 681

- mixed code set environments
    - application design 424
  - mixed Extended UNIX Code considerations 421
  - model for DB2 programming 21
  - modelling entities as independent objects 221
  - money using CREATE DISTINCT TYPE example 277
  - moving large objects using a file reference variable 227
  - multi-byte character support
    - code points for special characters 404
  - multi-byte code pages
    - Chinese (Traditional) code sets 419, 422
    - Japanese code sets 419, 422
  - multi-byte considerations
    - Chinese (Traditional) code sets in C/C++ 524
    - Chinese (Traditional) EUC code sets
      - in REXX 625
    - Japanese and traditional Chinese EUC code sets
      - in COBOL 590
      - in FORTRAN 603
    - Japanese code sets
      - in C/C++ 524
    - Japanese EUC code sets
      - in REXX 625
  - multiple definitions of SQLCA, avoiding 16
  - multiple threads
    - application dependencies
      - between contexts 444
    - database dependencies between contexts 444
    - guidelines 443
    - preventing deadlocks between contexts 445
    - using in DB2 applications 441
  - multiple triggers, ordering of 385
  - multisite update 433, 691
    - coding SQL for a multisite update application 435
    - configuration parameters 439
    - considerations with stored procedures 209
    - DB2 Connect support 691
    - general considerations 433
    - overview 434
    - restrictions 439
    - when to use 435
  - multisite update configuration parameter
    - LOCKTIMEOUT 439
    - RESYNC\_INTERVAL 439
    - SPM\_LOG\_NAME 440
    - SPM\_NAME 439
    - SPM\_RESYNC\_AGENT\_LIMIT 439
    - TM\_DATABASE 439
    - TP\_MON\_NAME 439
- ## N
- national language support
    - mixed-byte data 681
  - national language support (NLS)
    - character conversion 407
    - code page 407
    - considerations 394
  - nicknames
    - cataloging related information 472
    - considerations, restrictions 473
    - CREATE NICKNAME statement 473
    - using with views 475
  - NOCONVERT
    - WCHARTYPE
      - in stored procedures 209
  - NOLINEMACRO
    - PREP option 496
  - nonexecutable SQL statements
    - DECLARE CURSOR 16
    - INCLUDE 16
    - INCLUDE SQLDA 16
  - normal call, to a UDF 294
  - NOT ATOMIC compound SQL 690
  - NOT DETERMINISTIC option and UDFs 340
  - NOT FENCED LOB locator UDFs 335
  - NOT FENCED stored procedures
    - considerations 211
    - precompiling 211
    - working with 210
  - NOT NULL CALL clause 290
  - NOT NULL CALL option and UDFs 340
  - null-terminated character form
    - C/C++ type 525
  - null-terminator 530
  - NULL value
    - receiving, preparing for 73
  - numeric conversion overflows 688
  - numeric data types 681
  - numeric host variables
    - C/C++ 500
  - numeric host variables *(continued)*
    - COBOL 500
    - FORTRAN 597
  - NUMERIC parameter to UDF 304
  - NUMERIC SQL data type 320
    - C/C++ 525
    - COBOL 586
    - FORTRAN 603
    - Java 535
    - Java stored procedures (DB2GENERAL) 659
    - OLE DB table function 329
    - REXX 617
- ## O
- object, large data type 682
  - object instances
    - for OLE automation UDFs 319
  - Object Linking and Embedding (OLE) 317
  - object-orientation and UDFs 256
  - object oriented COBOL
    - restrictions 590
  - object-oriented extensions and UDTs 275
  - object-relational
    - application domain and object-orientation 221
    - constraint mechanisms 221
    - data types 221
    - definition 221
    - LOBs 221
    - support for 222
    - triggers 221
    - UDTs and UDFs 221
    - why use the DB2 object extensions 221
  - ODBC
    - access to data consideration 25
  - OLE automation data types 320
    - BSTR 320
    - DATE 320
    - double 320
    - float 320
    - long 320
    - SAFEARRAY 320
    - short 320
  - OLE automation object counting example 265
  - OLE automation server 318
  - OLE automation types 319
  - OLE automation types and BASIC types 320
  - OLE automation types and C++ types 320

- OLE automation UDFs
    - creatable multi-use OLE
      - automation server 323
    - creatable single-use OLE
      - automation server 323
    - implementation 318
    - implementation in BASIC 321
    - implementation in C++ 321
    - object instances 319
    - scratchpad considerations 319
    - UDFs 317
  - OLE DB
    - example
      - using connection string 326
      - using CONNECTSTRING
        - option 325
      - using server name 326
    - supported in DB2 26
    - table functions
      - CREATE SERVER
        - statement 328
      - CREATE USER MAPPING
        - statement 328
      - creating 325
      - defining a user mapping 328
      - EXTERNAL NAME
        - clause 327
      - fully qualified names 327
      - identifying servers 328
      - overview 324
  - OLE keyword 317
  - OLE messaging example 369
  - OLE programmatic ID (progID) 318
  - open state, buffered insert 459
  - OPENFTCH.SQB COBOL program
    - listing 97
  - OPENFTCH.SQC C program
    - listing 93
  - OPENFTCH.SQF FORTRAN
    - program listing 99
  - Openftch.sqlj Java program
    - listing 95
  - ORDER BY clause
    - sort order 686
  - OS/400 using DB2 Connect 679
  - OUT stored procedure
    - parameters 191, 196
  - Outcli.java Java program listing 204
  - output and input to screen and
    - keyboard and UDFs 342
  - output file extensions, C/C++
    - language 492
  - Outsrv.sqlj Java program listing 207
  - overloading
    - function names 258
  - overloading (*continued*)
    - stored procedure names 258
  - owner 684
- ## P
- package
    - attributes 684
    - creating 51
    - creating for compiled
      - applications 47
    - renaming 51
    - support to REXX
      - applications 620
    - timestamp errors 56
  - page-level locking 687
  - parameter markers 177
    - in functions example 269
  - in processing arbitrary
    - statements 157
  - programming example 167
  - SQLVAR entries 166
  - use in dynamic SQL 166
  - partitioned environment
    - error handling
      - considerations 467
    - identifying when errors
      - occur 469
    - improving performance 453
    - severe error considerations 467
  - pass-through
    - COMMIT statement 485, 486
    - considerations, restrictions 486
    - SET PASSTHRU RESET
      - statement 486
    - SET PASSTHRU statement 486
    - SQL processing 485
  - passing contexts between
    - threads 442
  - performance
    - dynamic SQL caching 60
    - factors affecting, static SQL 60
    - improving
      - in partitioned
        - environments 453
      - using buffered inserts 455
      - using directed DSS 454
      - using local bypass 454
      - using stored procedures 186
      - with READ ONLY
        - cursors 453
    - improving with read only
      - cursors 90
    - increasing using LOB
      - locators 335
    - large objects 229
  - performance (*continued*)
    - NOT FENCED stored
      - procedure 60
    - optimizing with packages 55
    - passing blocks of data 451
    - precompiling static SQL
      - statements 55
    - UDFs 256
  - performance advantages
    - with buffered insert 458
  - performance and UDTs 275
  - Perl
    - access to data consideration 26
  - PICTURE (PIC) clause in COBOL
    - types 586
  - portability 178
  - porting applications 679
  - precompile option
    - WCHARTYPE
      - NOCONVERT 211
  - precompiler
    - C/C++ language
      - #include macro 491
      - character set 491
      - debugging 496
      - macro processing 491
      - qualification and member
        - operators 518
      - symbol substitution 491
      - trigraph sequences 491
    - COBOL 569
    - DB2 Connect support 681, 683
    - FORTRAN 591
    - options 48
    - overview of 44
    - supported languages 10
    - types of output 48
  - precompiling 49
    - accessing DRDA application
      - server through DB2
        - Connect 49
    - accessing multiple servers 49
    - client application 210
    - example of 47
    - flagger utility 50
    - options, updatable cursor 91
    - overview of 47
    - supporting dynamic SQL
      - statements 129
  - PREP option
    - NOLINEMACRO 496
  - PREPARE statement
    - DB2 Connect support 693
    - processing arbitrary
      - statements 157



- PREPARE statement (*continued*)
  - summary of 693
- preprocessor functions and the SQL precompiler 510
- prerequisites, for programming 9
- primary key, DB2 Connect support 686
- printf() for debugging UDFs 371
- problem resolution
  - stored procedures 214
- processing SQL statements in REXX 609
- program variable type, data value control consideration 29
- programmable ID (progID) for OLE automation UDFs 318
- programming considerations
  - accessing host or AS/400 servers 441
  - collating sequences 394
  - conversion between different code pages 394
  - in a host or AS/400 environment 679
  - in C/C++ 491
  - in COBOL 569
  - in FORTRAN 591
  - in REXX 607
  - national language support 394
  - programming in complex environments 394
  - X/Open XA interface 448
- programming framework 21
- protecting UDFs 340
- prototyping SQL code 39
- PUT statement
  - not supported in DB2 Connect 693

## Q

- QSQ (DB2 Universal Database for AS/400) 683
- qualification and member operators in C/C++ 518
- qualifier
  - different platforms 684
- query products, access to data consideration 27
- QUERYOPT bind option 53
- Quick Beginnings for OS/2 724
- Quick Beginnings for UNIX 724
- Quick Beginnings for Windows NT 724

## R

- RAISE\_ERROR built-in function 383
- RDO specification
  - supported in DB2 26
- re-entrant
  - stored procedures 211
  - UDFs 331
- re-use and UDFs 256
- REAL\*2 FORTRAN type 603
- REAL\*4 FORTRAN type 603
- REAL\*8 FORTRAN type 603
- REAL parameter to UDF 305
- REAL SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- rebinding
  - description 57
  - REBIND PACKAGE command 57
  - REDEFINES in COBOL 585
  - REFERENCING clause in the CREATE TRIGGER statement 380
- referential integrity
  - data relationship consideration 30
  - DB2 Connect support 686
- referential integrity constraint
  - data value control consideration 28
- registering
  - OLE automation UDFs 318
  - UDFs 260
- registering Java stored procedures 557
- releasing your connection
  - CMS applications 18
  - to DB2 19
- Remote Data Object specification
  - supported in DB2 26
- remote unit of work 434
- renaming, package 51
- REORGANIZE TABLE
  - command 421
- repeatable read, technique for 101
- Replication Guide and Reference 723
- reporting errors 468
- restoring data 19
- restrictions
  - for UDFs 342
  - in C/C++ 510
  - in COBOL 569
  - in FORTRAN 591
  - in REXX 608
  - on using buffered inserts 460
- restrictions for DB2DARI stored procedures 657
- result code 15
- RESULT REXX predefined variable 612
- result sets
  - from stored procedures 212
- resume using CREATE DISTINCT TYPE example 277
- RESYNC\_INTERVAL multisite update configuration parameter 439
- retrieving
  - multiple rows 78
  - one row 62
  - rows 90
- retrieving data
  - a second time 102
  - scroll backwards, technique for 101
  - updating 104
- return code 15
  - SQLCA structure 117
- RETURNS clause in the CREATE FUNCTION statement 303
- RETURNS TABLE clause 289
- REVOKE statement
  - DB2 Connect support 682
- REXX
  - access to data consideration 24
  - API syntax 621
  - bind files
    - DB2ARXCS.BND REXX bind file 620
    - DB2ARXNC.BND REXX bind file 620
    - DB2ARXRR.BND REXX bind file 620
    - DB2ARXRS.BND REXX bind file 620
    - DB2ARXUR.BND REXX bind file 620
  - calling the DB2 CLP from application 621
  - Chinese (Traditional) considerations 625
  - clearing LOB host variables 617
  - cursor identifiers 610

REXX (*continued*)

- data types supported 24
- execution requirements 619
- indicator variables 612, 619
- initializing variables 623
- Japanese considerations 625
- LOB data 614
- LOB file reference
  - declarations 615
- LOB locator declarations 615
- predefined variables
  - RESULT 612
  - SQLCA 612
  - SQLISL 612
  - SQLMSG 612
  - SQLRDAT 612
  - SQLRIDA 612
  - SQLRODA 612
- programming
  - considerations 608
- registering routines in 608
- registering SQLEXEC, SQLDBS and SQLDB2 608
- restrictions in 608
- stored procedures in 623
- supported SQL statements 610

REXX and C++ data types 617

REXX APIs

- SQLDB2 607, 621
- SQLDBS 607
- SQLEXEC 607

ROLLBACK statement 11

- association with cursor 80
- backing out changes 19
- ending transactions 19
- in DB2 Connect 683
- restoring data 19
- rolling back changes 19

ROLLBACK WORK RELEASE

- not supported in DB2 Connect 693

rolling back changes 19

row

- order of, controlling 102
- order of in table, positioning 104
- retrieving multiple with cursor 90
- retrieving one 61
- retrieving using SQLDA 151
- selecting one, with SELECT INTO statement 62

row blocking

- customizing for performance 451

row id, data type 682

row-level locking 687

ROWID data type

- supported by DB2 Connect Version 6 682

rows

- set of rows affected by triggers 377

RQRIOLBK field 684

rules that govern operations on large objects 221

run-time services

- multiple threads, affect on latches 442

RUOW 434

## S

SAA1 530

SAFEARRAY OLE automation type 320

sales using CREATE TABLE example 277

sample programs 60

- Application Program Interface (API) 633
- embedded SQL statements 633
- Java stored procedures 557
- Java UDFs 312
- location of 633

savepoint, buffered insert consideration 456

scalar functions 259

- contents of call-type argument 295

schema-name and UDFs 258

scratchpad, passing to UDF 288, 293

scratchpad and UDFs 314, 331

SCRATCHPAD clause 296

scratchpad considerations

- for OLE automation UDFs 319

SCRATCHPAD keyword 293, 294, 314, 331

SCRATCHPAD option

- for OLE automation UDFs 319

section number 693

SELECT INTO statement

- overview of 62

SELECT statement

- association with EXECUTE statement 130
- buffered insert consideration 459
- DB2 Connect support 681
- declaring an SQLDA 148

SELECT statement (*continued*)

- describing, after allocating SQLDA 130
- in DECLARE CURSOR statement 79
- retrieving data a second time 102
- retrieving multiple rows with 78
- updating retrieved data 104
- varying-list, overview of 158

self-referencing tables 686

semantic behavior of stored objects 221

semaphores 444

serialization of data structures 443

serialization of SQL statement execution 442

server options 481

SET CURRENT FUNCTION PATH statement 260

SET CURRENT PACKAGESET statement 52

SET CURRENT statement

- DB2 Connect support 693

SET PASSTHRU RESET statement 486

SET PASSTHRU statement 486

SET SERVER OPTION statement 482

setAsciiStream JDBC API 566

setString JDBC API 566

setting up a DB2 program 11

setting up document server 729

setUnicodeStream JDBC API 566

severe errors

- considerations in a partitioned environment 467

shared memory size for UDFs 342

shift-out and shift-in characters 681

short C/C++ type 525

short int C/C++ type 525

short Java type 535

short OLE automation type 320

signal handlers

- installing, sample programs 104
- with SQL statements 119

SIGNAL SQLSTATE SQL statement to validate input data 374

signature, two functions and the same 258

SIGUSR1 interrupt 120

SIMPLE stored procedures 193

SIMPLE WITH NULLS stored procedures 193

SMALLINT 290

- SMALLINT parameter to UDF 304
- SMALLINT SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- snapshot monitor
  - diagnosing a suspended or looping application 469
- solving problems
  - numeric conversion overflows 688
- sort order 686
- sorting, specifying collating sequence 398
- source file
  - creating, overview of 45
  - file name extensions 48
  - modified source file, definition of 48
  - requirements 49
  - SQL file extensions 45
- source-level debuggers and UDFs 371
- sourced UDF 281
- special registers
  - CURRENT EXPLAIN MODE 52
  - CURRENT FUNCTION PATH 52
  - CURRENT QUERY OPTIMIZATION 52
- specific-name, passing to UDF 292
- SPM\_LOG\_SIZE multisite update configuration parameter 440
- SPM\_NAME multisite update configuration parameter 439
- SPM\_RESYNC\_AGENT\_LIMIT multisite update configuration parameter 439
- SQL
  - authorization considerations 32
  - authorization considerations for dynamic SQL 33
  - authorization considerations for static SQL 33
  - authorization considerations using APIs 34
  - dynamically prepared 178
- SQL\_API\_FN macro 303, 656
- SQL-argument 295
- SQL-argument, passing to UDF 289
- SQL-argument-ind 295
- SQL-argument-ind, passing to UDF 290
- SQL arguments, passing from DB2 to a UDF 288
- SQL Communications Area (SQLCA) 14
- SQL data types 319, 320
- SQL Data Types
  - BIGINT 75
  - BLOB 75, 320
  - CHAR 75, 320
  - CLOB 75, 320
  - COBOL 586
  - conversion to C/C++ 525
  - DATE 75, 320
  - DBCLOB 75, 320
  - DECIMAL 75
  - DOUBLE 320
  - FLOAT 75, 320
  - FOR BIT DATA 320
  - FORTTRAN 603
  - GRAPHIC 320
  - INTEGER 75, 320
  - Java 535
  - LONG GRAPHIC 320
  - LONG VARCHAR 75, 320
  - LONG VARGRAPHIC 75, 320
  - NUMERIC 320
  - OLE DB table function 329
  - REAL 75, 320
  - REXX 617
  - SMALLINT 75, 320
  - TIME 75, 320
  - TIMESTAMP 75, 320
  - VARCHAR 75, 320
  - VARGRAPHIC 75, 320
- SQL data types, passed to a UDF 303
- SQL declare section 11
- SQL/DS using DB2 Connect 679
- SQL\_FILE\_READ, input value option 246
- SQL Getting Started 723
- SQL include file
  - for C/C++ applications 493
  - for COBOL applications 570
  - for FORTRAN applications 592
- SQL Reference 723
- SQL-result 295, 333
- SQL-result, passing to UDF 289
- SQL-result-ind 295, 333
- SQL-result-ind, passing to UDF 290
- SQL-state, passing to UDF 291
- SQL statement categories 679
- SQL statement execution
  - serialization 442
- SQL statements
  - C/C++ syntax 497
  - COBOL syntax 573
  - DB2 Connect support 692, 693
  - exception handlers 120
  - FORTTRAN syntax 595
  - grouping using stored procedures 187
  - interrupt handlers 120
  - REXX syntax 609
  - saving end user requests 158
  - signal handlers 120
  - supported in REXX 610
- SQL\_WCHART\_CONVERT preprocessor macro 521
- SQL1252A include file
  - for COBOL applications 572
  - for FORTRAN applications 594
- SQL1252B include file
  - for COBOL applications 572
  - for FORTRAN applications 594
- SQL92 686
- SQLADEF include file
  - for C/C++ applications 493
- SQLAPREP include file
  - for C/C++ applications 493
  - for COBOL applications 570
  - for FORTRAN applications 592
- SQLCA
  - avoiding multiple definitions 16
  - error reporting in buffered insert 459
  - incomplete insert when error occurs 459
  - multithreading considerations 443
  - SQLERRMC field 683, 690
  - SQLERRP field 682
- SQLCA\_92 include file
  - for COBOL applications 570
  - for FORTRAN applications 593
- SQLCA\_92 structure
  - include file for FORTRAN applications 593
- SQLCA\_CN include file 592
- SQLCA\_CS include file 592
- SQLCA include file
  - for C/C++ applications 493
  - for COBOL applications 570
  - for FORTRAN applications 592
- SQLCA predefined variable 612



- SQLCA.SQLERRD settings on CONNECT 427
- SQLCA structure
  - defining, sample programs 104
  - include file
    - for C/C++ applications 493
    - for COBOL applications 570
    - for FORTRAN applications 592
  - merged multiple structures 468
  - overview 117
  - reporting errors 468
  - requirements 117
  - sqlerrd 468
  - SQLERRD(6) field 469
  - SQLWARN1 field 74
  - using in stored procedures 658
  - warnings 74
- SQLCHAR structure
  - passing data with 156
- SQLCLI include file
  - for C/C++ applications 493
- SQLCLI include file
  - for C/C++ applications 493
- SQLCODE
  - in Java programs 537
  - including SQLCA 15
  - platform differences 687
  - reporting errors 468
  - standalone 686
  - structure 117
- SQLCODE -1015 467
- SQLCODE -1034 467
- SQLCODE -1224 468
- SQLCODES include file
  - for C/C++ applications 493
  - for COBOL applications 570
  - for FORTRAN applications 593
- SQLDA
  - multithreading considerations 443
- SQLDA include file
  - for C/C++ applications 494
  - for COBOL applications 571
  - for FORTRAN applications 593
- sqllda.n.SQLDAT 657
- sqllda.n.SQLDATALEN 657
- sqllda.n.SQLDATATYPE\_NAME 657
- sqllda.n.SQLLIND 657
- sqllda.n.SQLEN 657
- sqllda.n.SQLLONGLEN 657
- sqllda.n.SQlname.data 657
- sqllda.n.SQlname.length 657
- sqllda.n.SQLTYPE 657
- sqllda.SQDABC 657
- sqllda.SQDAID 657
- sqllda.SQLN 657
- SQLDA structure
  - association with PREPARE statement 130
  - creating (allocating) 152
  - creating, host language examples 153
  - declaring 148
  - declaring sufficient SQLVAR entities 150
  - fields used in stored procedures
    - SQLDATA 658
    - SQLIND 658
    - SQLEN 658
    - SQLTYPE 658
  - initializing for stored procedure (DB2DARI) 655
  - input-SQLDA procedure, sample of 668
  - manipulation with DB2DARI stored procedure 657
  - passing data with 156
  - preparing statements using minimum 149
  - server input procedure, sample of 675
  - used to pass blocks of data 452
  - using, sample program 159
  - using in stored procedures 658
- SQLDACT include file 593
- SQLDATA field 658
- SQLDB2, registering for REXX 608
- SQLDB2 REXX API 607, 621
- sqldbchar and wchar\_t, selecting data types 520
- sqldbchar C/C++ type 525
- sqldbchar data type 307, 308, 310, 520
- SQLDBS, registering for REXX 608
- SQLDBS REXX API 607
- SQLE819A include file
  - for C/C++ applications 494
  - for COBOL applications 571
  - for FORTRAN applications 593
- SQLE819B include file
  - for C/C++ applications 494
  - for COBOL applications 571
  - for FORTRAN applications 594
- SQLE850A include file
  - for COBOL applications 571
  - for FORTRAN applications 594
- SQLE850B include file
  - for COBOL applications 571
  - for FORTRAN applications 594
- SQLE859A include file
  - for C/C++ applications 494
- SQLE859B include file
  - for C/C++ applications 494
- SQLE932A include file
  - for C/C++ applications 494
  - for COBOL applications 572
  - for FORTRAN applications 594
- SQLE932B include file
  - for C/C++ applications 495
  - for COBOL applications 572
  - for FORTRAN applications 594
- sqlAttachToCtx() API 442
- SQLCAU include file
  - for C/C++ applications 494
  - for COBOL applications 571
  - for FORTRAN applications 593
- sqlBeginCtx() API 442
- sqlDetachFromCtx() API 442
- sqlEndCtx() API 442
- sqlGetCurrentCtx() API 442
- sqlInterruptCtx() API 442
- SQLENV include file
  - for C/C++ applications 494
  - for COBOL applications 571
  - for FORTRAN applications 593
- SQLERRD(1) 409, 425, 427
- SQLERRD(2) 409, 425, 427
- SQLERRD(3)
  - in an XA environment 450
- SQLERRMC field of SQLCA 409, 683, 690
- SQLERRP field of SQLCA 682
- sqlSetTypeCtx() API 442
- SQLETS include file
  - for COBOL applications 571
- SQLException
  - handling 123
  - retrieving SQLCODE from 537
  - retrieving SQLSTATE from 537
- SQLEXEC
  - processing SQL statements in REXX 609
- SQLEXEC, registering for REXX 608
- SQLEXEC REXX API 607
- SQLEXT include file
  - for CLI applications 494
- SQLIND field 658
- SQLISL predefined variable 612
- SQLJ (Embedded SQL for Java)
  - applets 547
  - calling stored procedures 554
  - db2prof 546
  - db2profp 546

- SQLJ (Embedded SQL for Java)
  - (continued)
  - declaring cursors 547
  - declaring iterators 549
  - embedding SQL statements
    - in 548
  - example clauses 548
  - example program using 550
  - host variables 554
  - Java Database Connectivity (JDBC) interoperability 567
  - overview 545
  - positioned DELETE
    - statement 549
  - positioned UPDATE
    - statement 549
  - profconv 546
  - restrictions 547
  - translator 546
- SQLJ (Embedded SQLJ for Java)
  - comparison with Java Database Connectivity (JDBC) 533
- SQLJACB include file
  - for C/C++ applications 495
- SQLLEN field 658
- SQLMON include file
  - for C/C++ applications 495
  - for COBOL applications 572
  - for FORTRAN applications 594
- SQLMONCT include file
  - for COBOL applications 572
- SQLMSG predefined variable 612
- SQLRDAT predefined variable 612
- SQLRIDA predefined variable 612
- SQLRODA predefined variable 612
- SQLSTATE
  - differences 687
  - in CLI 177
  - in Java programs 537
  - in SQLERRMC field of SQLCA 690
  - standalone 686
- SQLSTATE field, in error messages 117
- SQLSTATE include file
  - for C/C++ applications 495
  - for COBOL applications 572
  - for FORTRAN applications 595
- SQLSYSTEM include file
  - for C/C++ applications 495
- SQLTYPE field 658
- sqludf\_append API 335
- sqludf\_create\_locator API 335
- sqludf\_free\_locator API 335
- sqludf.h include file 303
- sqludf.h include file for UDFs 311
- SQLUDF include file
  - description 311
  - for C/C++ applications 495
  - UDF interface 288
- sqludf\_length API 335
- sqludf\_substr API 335
- SQLUTBCQ include file
  - for COBOL applications 572
- SQLUTBSQ include file
  - for COBOL applications 572
- SQLUTIL include file
  - for C/C++ applications 495
  - for COBOL applications 573
  - for FORTRAN applications 595
- SQLUV include file
  - for C/C++ applications 495
- SQLUVEND include file
  - for C/C++ applications 495
- SQLVAR entities
  - declaring sufficient number of 150
  - variable number of, declaring 148
- SQLWARN structure, overview of 117
- SQLXA include file
  - for C/C++ applications 495
- SQLZ\_DISCONNECT\_PROC return value 199
- SQLZ\_HOLD\_PROC return value 199
- statement handle 177
- statements
  - BEGIN DECLARE SECTION 11
  - COMMIT 18
  - CONNECT 17
  - END DECLARE SECTION 11
  - INCLUDE SQLCA 14
  - ROLLBACK 19
- STATIC.SQB COBOL program listing 66
- STATIC.SQC C program listing 64
- STATIC.SQF FORTRAN program listing 67
- static SQL
  - coding statements to retrieve and manipulate data 68
  - comparing to dynamic SQL 130
  - considerations 130
  - DB2 Connect support 679
  - overview 60
  - precompiling, advantages of 55
  - sample program 61
- static SQL (continued)
  - static update programming
    - example 68
    - using host variables 69
- Static.sqlj Java program listing 65
- storage
  - allocating to hold row 151
  - declaring sufficient SQLVAR entities 148
- storage allocation for unequal code pages 425
- Stored Procedure Builder
  - debug table 560
  - environment settings 559
- stored procedures 689
  - advantages 186
  - allocating storage 190
  - allowed SQL statements in 197
  - application logic
    - consideration 31
  - application troubleshooting 214
  - architecture 188
  - building an application 209
  - C++ consideration 208
  - CALL statement 190
  - character conversion 405
  - character conversions, for EUC 432
  - Chinese (Traditional) code sets 424
  - client application 190
  - code page considerations 208
  - compiling and linking 209
  - CONTAINS SQL clause 197
  - CREATE PROCEDURE
    - statement 190
  - creating and using in Java 557
  - db2dari executable 198
  - debugging 210
    - using Stored Procedure Builder 559
  - declaring parameter modes 191
  - example 196
  - EXTERNAL clause 191
  - FOR BIT DATA
    - considerations 208
  - graphic host variable
    - considerations 209
  - host variables 189
  - initializing REXX variables 623
  - input-SQLDA procedure, sample of 668
  - input-SQLDA stored procedure, sample of 675
  - invoking 190

- stored procedures 186 (*continued*)
    - Japanese code sets 186
    - LANGUAGE clause 193
    - languages supported 193
    - location 191
    - multisite update
      - considerations 209
    - NOT FENCED 210
    - OUT host variable, sample 200
    - overloading names 191
    - overview of 185, 190
    - PARAMETER STYLE clause 193
    - parameters 191, 196, 197
    - passing DBINFO structures 195
    - path 191
    - PROGRAM TYPE clause 196
    - registering in Java 557
    - registering with CREATE PROCEDURE 190
    - requirements for 188
    - restrictions 198
    - return values 199
    - returning result sets 212
    - REXX applications 623
    - sample OUT client program 202
    - using the SQLDA and SQLCA structures 658
    - where to place 191
    - written as a main function 196
  - stored procedures (DB2DARI)
    - calling convention
      - parameter conversion 656
      - SQL\_API\_FN 656
    - data structure usage 657
    - NOT FENCED 667
    - parameter variables
      - sqllda.n.SQLDAT 657
      - sqllda.n.SQLDATALEN 657
      - sqllda.n.SQLDATATYPE\_NAME 657
      - sqllda.n.SQLIND 657
      - sqllda.n.SQLLEN 657
      - sqllda.n.SQLLONGLEN 657
      - sqllda.n.SQLNAME.data 657
      - sqllda.n.SQLNAME.length 657
      - sqllda.n.SQLTYPE 657
      - sqllda.SQLDABC 657
      - sqllda.SQLDAID 657
      - sqllda.SQLN 657
    - restrictions 657
    - using indicator variables 657
  - storing large objects 221
  - String 535
  - string search and defining UDFs
    - example 262
  - string search on BLOBs 263
  - string search over UDT
    - example 263
  - strong typing and UDTs 279
  - structure definitions in sqludf.h 311
  - Structured Query Language (SQL)
    - statements, summary 32
    - table of supported statements 629
  - success code 15
  - surrogate functions 374
  - suspended application
    - diagnosing 469
  - symbol substitutions, C/C++
    - language restrictions 510
  - Sync Point Manager 440
  - syntax
    - character host variables 502
    - declare section
      - in C/C++ 499
      - in COBOL 575
      - in FORTRAN 597
    - embedded SQL statement
      - avoiding line breaks 498
      - comments in C/C++ 497
      - comments in COBOL 573
      - comments in FORTRAN 596
      - comments in REXX 611
      - in C/C++ 497
      - in COBOL 573
      - in FORTRAN 595
      - substitution of white space characters 498
    - graphic host variable
      - in C/C++ 503
    - processing SQL statement
      - in REXX 609
  - syntax for referring to
    - functions 267
  - SYSCAT.FUNCMAPOPTIONS
    - catalog view 483
  - SYSCAT.FUNCTIONS catalog
    - view 483
  - SYSIBM.SYSPROCEDURES catalog
    - (OS/390) 689
  - SYSSTAT.FUNCTIONS catalog
    - view 483
  - system catalog
    - DB2 Connect support 687
  - system catalog views
    - prototyping utility 40
  - system configuration parameter for
    - shared memory size 342
  - System.err Java I/O stream 313
  - System.in Java I/O stream 313
  - System Monitor Guide and Reference 723
  - System.out Java I/O stream 313
- ## T
- table
    - committing changes 18
    - data source tables 472
    - lob-options-clause of the CREATE TABLE statement 229
    - positioning cursor at end 104
    - tablespace-options-clause of the CREATE TABLE statement 229
  - table check constraint, data value
    - control consideration 28
  - table function 289
    - SQL-result argument 289
  - table function example 266
  - table functions 259
    - application design
      - considerations 333
    - contents of call-type argument 296
    - in Java 313
    - OLE DB 324
  - table names
    - resolving unqualified 52
  - tablespace-options-clause of the CREATE TABLE statement 229
  - target partition
    - behavior without buffered insert 457
  - territory, in SQLERRMC field of SQLCA 683
  - test data
    - generating 37
  - test database
    - CREATE DATABASE API 36
    - creating 35
    - recommendations 36
  - testing and debugging utilities
    - database system monitor 39
    - Explain facility 39
    - flagger 39
    - system catalog views 39
    - updating system catalog statistics 39
  - testing environment
    - for partitioned environments 466
    - setting up 35
    - test databases, guidelines for creating 35
  - testing your UDF 371
  - tfweather\_u table function C
    - program listing 355

- TIME parameter to UDF 309
- TIME SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- timeout on a lock 687
- TIMESTAMP parameter to UDF 309
- TIMESTAMP SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- TM\_DATABASE multisite update configuration parameter 439
- TP\_MON\_NAME multisite update configuration parameter 439
- transaction
  - beginning a transaction 18
  - committing work done 18
  - description 17
  - ending a transaction 18
  - ending the program
    - COMMIT and ROLLBACK statements 19
  - implicitly ending 20
  - rolling back work done 19
- transaction log, buffered insert consideration 458
- transaction log consideration for buffered insert 458
- transaction processing monitors
  - X/Open XA Interface 448
- transition tables
  - based on type of trigger
    - event 380
  - OLD\_TABLE and NEW\_TABLE 380
- transition variables, OLD and NEW
  - transition variables based on type of trigger event 379
- translation, character 681
- trigger event such as UPDATE, INSERT or DELETE 376
- triggers
  - activation time 378
  - triggers (*continued*)
    - AFTER triggers 378, 383
    - and DB2 object extensions 221
    - application logic
      - consideration 32
    - BEFORE triggers 378, 383
    - benefits 374
    - cascading 385
    - data relationship
      - consideration 30
    - definition 373
    - DELETE operation 376
    - functions with SQL triggered statements
      - RAISE\_ERROR built-in function 383
    - INSERT operation 376
    - interactions with referential constraints 385
    - multiple triggers, ordering of 385
    - overview 375
    - referential constraints, interactions with using triggers 385
    - set of affected rows 377
    - synergy with UDTs, UDFs, and LOBs 386
    - transition tables 380
    - transition variables 379
    - trigger event 376
    - trigger granularity 377
    - triggered action condition 382
    - triggered SQL statement 382
    - UPDATE operation 376
    - WHEN clause 382
    - why use triggers 373
  - trigraph sequences 491
  - troubleshooting
    - stored procedures 214
    - using Visual Studio 214
  - Troubleshooting Guide 724
  - truncation
    - host variables 74
    - indicator variables 74
  - type conversion
    - between SQL types and OLE automation types 319
  - type decoration
    - in stored procedures 208
    - in UDFs 343
  - type decoration consideration
    - C++ 492
- type mapping
  - OLE automation types and BASIC types 320
  - OLE automation types and C++ types 320
- types, abstract 682
- types, user defined 682
- types, user defined distinct 682
- types or arguments, promotions in UDFs 303

## U

- UCS-2 419
- UDFs (User-defined functions)
  - allocating dynamic memory in the UDF 340
  - and DB2 object extensions 221
  - C++ consideration 343
  - calling convention 303
  - casting 273
  - caveats 342
  - Chinese (Traditional) code sets 423
  - code page differences 342
  - coding in Java 313
  - concepts 258
  - considerations when using protected resources 342
  - creating and using in Java 312
  - db2udf executable 342
  - debugging your UDF 371
  - definition 255
  - DETERMINISTIC 332
  - example 348
  - examples of UDF code 344
  - EXTERNAL ACTION
    - option 340
  - FENCED option 340
  - function path 258
  - function selection algorithm 258
  - general considerations 271
  - hints and tips for coding 340
  - implementing UDFs 256
  - infix notation 271
  - input and output to screen and keyboard 342
  - interface between DB2 and a UDF 288
  - invoking
    - examples of invocations 267
    - parameter markers in functions 269
    - qualified function
      - reference 269
    - unqualified function
      - reference 270

- UDFs (User-defined functions)
  - (continued)
  - Japanese code sets 340
  - Java consideration 288
  - list of types and their representations in UDFs 303
  - LOB locator usage scenarios 339
  - LOB types 272
  - NOT DETERMINISTIC 331
  - NOT DETERMINISTIC option 340
  - NOT FENCED 346
  - NOT NULL CALL 346
  - NOT NULL CALL option 340
  - OLE automation UDFs 317
  - output and input to screen and keyboard 342
  - overloading function names 258
  - passing arguments from DB2 to a UDF 288
  - process of implementation 260
  - re-entrant UDFs 331
  - referring to functions 267
  - registering UDFs 261
    - examples of registering 261
  - restrictions and caveats 342
  - save state in function 331
  - schema-name and UDFs 258
  - SCRATCHPAD 332
  - scratchpad considerations 331
  - shared memory size 342
  - sourced 281
  - SQL\_API\_FN 345
  - SQL data types, how they are passed 303
  - SQLUDF include file 288, 311
  - SUBSTR built-in function 351
  - summary of function references 271
- UDFs (User-defined Functions)
  - synergy with triggers, UDTs, and LOBs 386
- UDFs (User-defined functions)
  - synergy with UDTs and LOBs
    - examples of complex applications 223
  - system configuration parameter for shared memory size 342
  - table functions 333
  - type of functions 259
  - unqualified reference 258
  - using LOB locators 335
  - why use UDFs 256
  - writing UDFs 261
  - writing your own UDF 287
- UDFs and LOB types 272
- UDTs (User-defined types)
  - and DB2 object extensions 221
  - defining a UDT 276
  - defining tables 277
  - manipulating
    - examples of 279
  - resolving unqualified UDTs 276
  - strong typing 279
- UDTs (User-defined Types)
  - synergy with triggers, UDFs, and LOBs 386
- UDTs (User-defined types)
  - synergy with UDFs and LOBs
    - examples of complex applications 223
  - UDTs type 682
- unambiguous cursors 684
- unequal code pages 425
  - allocating storage 425
- unfenced stored procedures 210
- Unicode (UCS-2)
  - character conversion 433
  - character conversion overflow 431
  - Chinese (Traditional) code sets 419
  - Japanese code sets 419
  - UDF considerations 423
- unique key violation
  - buffered insert 459
- unit of work
  - completing 80
  - cursor considerations 80
  - distributed 433
  - remote 434
- unqualified function reference
  - example 270
- unqualified reference 258
- unqualified table names
  - resolving 52
- UPDAT.CMD REXX program
  - listing 115
- UPDAT.SQB COBOL program
  - listing 111
- UPDAT.SQC C program listing 107
- UPDAT.SQF FORTRAN program
  - listing 113
- Updat.sqlj Java program listing 109
- update operation 375
- UPDATE operation and triggers 376
- UPDATE statement
  - DB2 Connect support 681
- USAGE clause in COBOL types 586
- use of UDTs in UNION
  - example 284
- user-defined collating
  - sequence 686, 711
- user-defined function, application
  - logic consideration 31
- user-defined sourced functions on UDTs example 282
- User Defined Types
  - supported by DB2 Connect 682
- USER MAPPING in OLE DB table
  - functions 328
- user updatable catalog statistics
  - prototyping utility 40
- using
  - Java stored procedures 557
  - Java UDFs 312
- using a locator to work with a CLOB
  - value example 231
- using qualified function reference
  - example 269
- UTIL.C program listing 120
- UTIL.F program listing 126
- utility APIs
  - include file for C/C++
    - applications 495
  - include file for COBOL
    - applications 571, 572, 573
  - include file for FORTRAN
    - applications 595

## V

- VALIDATE RUN
  - DB2 Connect support 683
- VALUES clause
  - on INSERT statement 457
- VARCHAR 292, 293
- VARCHAR FOR BIT DATA
  - parameter to UDF 306
- VARCHAR SQL data type 75, 320
  - C/C++ 525
  - C or C++ 530
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- VARCHAR structured form C/C++
  - type 525
- VARGRAPHIC data 530
- VARGRAPHIC parameter to UDF 308



- VARGRAPHIC SQL data type 75, 320
  - C/C++ 525
  - COBOL 586
  - FORTRAN 603
  - Java 535
  - Java stored procedures (DB2GENERAL) 659
  - OLE DB table function 329
  - REXX 617
- variable-length strings 681
- variables
  - SQLCODE 530, 589, 604
  - SQLSTATE 530, 589, 604
- variables, declaring 11
- variables, predefined in REXX 612
- Varinp.java Java program
  - listing 171
- VARINP.SQB COBOL program
  - listing 173
- VARINP.SQC C program listing 169
- VARINP.SQF FORTRAN program
  - listing 175
- view
  - data source views 472
  - data value control
    - consideration 29
  - DB2 Connect support 687
- Visual Basic
  - supported in DB2 26
- Visual C++
  - supported in DB2 26

**W**

- warning message, truncation 74
- wchar\_t and sqldbchar, selecting data types 520
- wchar\_t C/C++ type 525
- wchar\_t data type 307, 308, 310, 520
- WCHARTYPE
  - guidelines 522
  - in stored procedures 209
- WCHARTYPE precompiler
  - option 211, 521
- weight, definition of 394
- What's New 724
- WHENEVER SQLERROR
  - CONTINUE statement 16
- WHENEVER statement
  - caution in using with SQL statements 16
  - error indicators with SQLCA 15
  - in error handling 118
- wild moves, DB2 checking of 371
- Windows 95 code pages 401
  - DB2CODEPAGE environment variable 401
  - supported code pages 401
- Windows NT code pages 401
  - DB2CODEPAGE environment variable 401
  - supported code pages 401
- Windows registration database
  - for OLE automation UDFs 318
- work environment
  - setting up 35
  - test databases, guidelines for creating 35

## X

- X/Open XA Interface 448
  - API restrictions 451
  - characteristics of transaction processing 448
  - CICS environment 448
  - COMMIT and ROLLBACK 449
  - cursors declared WITH HOLD 450
  - DISCONNECT 449
  - multithreaded application 451
  - SET CONNECTION 449
  - single-threaded application 451
  - SQL CONNECT 449
  - transactions 448
  - XA environment 450
  - XASerialize 451

---

## Contacting IBM

This section lists ways you can get more information from IBM.

If you have a technical problem, please take the time to review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. Depending on the nature of your problem or concern, this guide will suggest information you can gather to help us to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

### Telephone

If you live in the U.S.A., call one of the following numbers:

- 1-800-237-5511 to learn about available service options.
- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, see Appendix A of the IBM Software Support Handbook. You can access this document by accessing the following page:

<http://www.ibm.com/support/>

then performing a search using the keyword “handbook”.

Note that in some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

### World Wide Web

<http://www.software.ibm.com/data/>

<http://www.software.ibm.com/data/db2/library/>

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more. The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information. (Note that this information may be in English only.)

### Anonymous FTP Sites

<ftp.software.ibm.com>

Log on as anonymous. In the directory /ps/products/db2, you can find demos, fixes, information, and tools concerning DB2 and many related products.

**Internet Newsgroups**

comp.databases.ibm-db2, bit.listserv.db2-l

These newsgroups are available for users to discuss their experiences with DB2 products.

**CompuServe**

**GO IBMDB2** to access the IBM DB2 Family forums

All DB2 products are supported through these forums.

To find out about the IBM Professional Certification Program for DB2 Universal Database, go to <a href="http://www.software.ibm.com/data/db2/db2tech/db2cert.html">http://www.software.ibm.com/data/db2/db2tech/db2cert.html</a>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------







Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SC09-2845-00

