

DB2® Server for VSE



Application Programming

Version 6 Release 1

DB2® Server for VSE



Application Programming

Version 6 Release 1

Note!

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

This book is also provided as an online book that can be viewed with the IBM® BookManager® READ and IBM Library Reader™ licensed programs.

First Edition (December 1998)

This edition, SC09-2661, applies to Version 6 Release 1 of the IBM DATABASE 2™ Server for VSE & VM Program 5648-A70, and to all subsequent releases of this product until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to: IBM Canada Ltd. Laboratory Information Development 2G/345/1150/TOR 1150 Eglinton Ave East, North York, Ontario, Canada M3C 1H7.

You can also send your comments by facsimile to (416) 448-6161 addressed to the attention of the RCF Coordinator. If you have access to Internet, you can send your comments electronically to **torrcf@ca.ibm.com**; IBMLink™, to **toribm(torrcf)**; IBM/PROFS®, to **torolab4(torrcf)**; IBMMAIL, to **ibmmail(caibmwt9)**; or through our home page at **<http://www.software.ibm.com/data/db2/vse-vm/>**

If you choose to respond through Internet, please include either your entire Internet network address, or a postal address.

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 1987, 1998. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix	Creating the Prolog	6
Programming Interface Information	ix	Creating the Body	10
Trademarks	ix	Creating the Epilog	12
About This Manual	xi	Using Logical Units of Work	14
Audience and Purpose of This Book	xi	Defining the Logical Unit of Work	14
Organization of This Book	xi	Beginning a Logical Unit of Work	14
Related Publications	xiii	Considering the CICS/VSE Logical Unit of Work	15
Syntax Notation Conventions	xiii	Ending a Logical Unit of Work	15
SQL Reserved Words	xvi	Summary	16
Conventions for Representing DBCS Characters	xvii	Using Host-Dependent Sample Applications	17
Components of the Relational Database Management System	xviii	Chapter 3. Coding the Body of a Program	21
Summary of Changes for DB2 Version 6 Release 1	xix	Contents	21
Enhancements, New Functions, and New Capabilities	xix	Defining Static SQL Statements	24
DRDA® RUOW Application Requestor for VSE (Online)	xix	Naming Conventions	24
Stored Procedures	xix	Coding SQL Statements to Retrieve and Manipulate Data	25
TCP/IP Support for DB2 Server for VM	xx	Retrieving Data	25
New Code Page and Euro Symbol Code Page Support	xx	Defining an SQL Query	25
DataPropagator™ Capture	xxi	Retrieving or Inserting Multiple Rows	32
QMF for VM, QMF for VSE, and QMF for Windows®	xxi	Retrieving Single Rows	37
RDS Above the Line	xxi	Constructing Search Conditions	38
Combining of NLS Feature Installation Tapes with Base Product Installation Tape	xxi	Performing Arithmetic Operations	38
Control Center Feature	xxii	Using Null Values	40
Data Restore Feature	xxii	Using the Predicates of a Search Condition	40
DB2 REXX SQL Feature	xxii	Using Functions	42
Reliability, Availability, and Serviceability Improvements	xxii	Using Column Functions	42
Migration Considerations	xxiii	Using Scalar Functions	43
Library Enhancements	xxiii	Using Data Types	43
Chapter 1. Getting Started	1	Assigning Data Types When the Column Is Created	43
Contents	1	Using Long Strings	45
What is the DB2 Server for VSE Product?	2	Using Datetime Data Types	45
What is SQL?	3	Using Character Subtypes and CCSIDs	46
Embedding SQL Statements in Host Language Programs	3	Converting Data	48
Writing a Program	4	Truncating Data	49
Chapter 2. Designing a Program	5	Using a Double-Byte Character Set (DBCS)	50
Contents	5	Using Expressions	51
Defining the Main Parts of a Program	6	Using Arithmetic Operators	52
		Using Special Registers	53
		Concatenating Character and Graphic Strings	54
		Using Host Variables	54
		Using Host Structures	55
		Using Constants	55
		Using Indicator Variables	59
		Using Views	61
		Creating a View	62
		Querying Tables through a View	63

Using Views to Manipulate Data	64	Coding Examples	103
Dropping a View	66	Special Considerations for C	103
Joining Tables	66	Special Considerations for PL/I	103
Joining Tables Using the Database		Result Sets	104
Manager	66		
Performing a Simple Join Query	66	Chapter 4. Preprocessing and Running a	
Joining Another User's Tables	67	Program	105
Analyzing How a Join Works	68	Contents	105
Using VARCHAR and VARGRAPHIC within		Defining the Steps to Execute the Program .	106
Join Conditions	68	Using 31-Bit Addressing	106
Using Nulls within Join Conditions	68	How DB2 Establishes User IDs for	
Joining a Table to Itself Using a Correlation		CICS/VSE Transactions	107
Name	69	User IDs for Remote CICS/VSE	
Imposing Limits on Join Queries	71	Transactions	108
Using SELECT * In a Join	71	Preprocessing the Program	108
Grouping the Rows of a Table	71	Preprocessing by Mode	109
Using VARCHAR and VARGRAPHIC within		Defining the Preprocessing Parameters .	112
Groups	72	Using the Preprocessor Option Member	121
Using Nulls within Groups	73	Using the Flagger at Preprocessor Time .	122
Using Select-Lists in Grouped Queries . . .	73	Using the CICS/VSE Translator	123
Using a WHERE Clause with a GROUP BY		Improving Performance Using	
Clause	73	Preprocessing Parameters	124
Using the HAVING Clause	73	Using the INCLUDE Statement	131
Combining Joins	74	Compiling the Program	132
Illustrating Grouping with an Exercise . .	74	Link-Editing and Loading the Program	132
Nesting Queries	77	Link-Editing the Program with	
Using the IN Predicate with a Subquery .	80	Supplementary Information	132
Considering Other Subquery Issues	81	Running the Program	134
Executing Subqueries Repeatedly:		Using a Consistency Token	134
Correlation	81	Loading the Package and Rebinding . . .	134
Writing a Correlated Subquery	83	Running by Mode	135
How the Database Manager Does		Running under CICS/VSE Support	136
Correlation	83	Accessing Other DB2 Family Application	
Illustrating a Correlated Subquery	85	Servers	136
Using a Subquery to Test for the Existence		Installing Applications that Access the	
of a Row	90	Database Manager	136
Table Designation Rule for Correlated		Installing a Batch Application	136
Subqueries	90	Installing an Online CICS/VSE Application	137
Combining Queries into a Single Query:		Distributing Packages across Like and	
UNION	91	Unlike Systems	139
SQL Comments within Static SQL Statements	94	Creating a Package Using CBND	140
Using Stored Procedures	95		
Writing Stored Procedures	97	Chapter 5. Testing and Debugging	145
Returning Information from the SQLCA . . .	98	Contents	145
Language Environment® (LE)		Doing Your Own Testing	146
Considerations	98	Checking Warnings and Errors at	
Preparing to Run a Stored Procedure . .	98	Preprocessor Time	146
Calling Stored Procedures	99	Testing SQL Statements	147
Authorization	100	Using the Automatic Error-Handling Facilities	147
AUTHIDs	100	Using the SQLCA	148
Stored Procedure Parameters	100	Examining Errors	150
Datatype Compatibility	101	Handling Errors in a Select-List	155
Conventions for Passing Stored Procedure		Handling Arithmetic Errors	155
Parameters	101	Handling Numeric Conversion Errors . . .	156

Handling CCSID Conversion Errors	157	Automatically Locking Dbspaces	204
Chapter 6. Using Dynamic Statements	159	Overriding Automatic Locking	205
Contents	159	Deleting the Contents of Dbspaces	206
Dynamically Defining SQL Statements	160	Other Data Definition Statements	206
Comparing Non-Query Statements to Query		Using Tables, Indexes, Statistics,	
Statements	160	Synonyms, Comments, and Labels	207
Using Non-Query Statements	160	Using Stored Procedures and PSERVERS	209
Executing Non-Parameterized Statements	161	Chapter 9. Assigning Authority and	
Executing Parameterized Statements	162	Privileges	213
Using Query Statements	164	Contents	213
Executing a Non-Parameterized		Defining User Access to the Database	214
Select-Statement	164	Defining Authority Types for the Database	214
Executing a Parameterized SELECT		Granting Authority to Users	214
Statement	171	Revoking Authority from Users	214
Executing a Parameterized Non-Query		Defining Privileges	215
Statement	175	Defining Privileges on Tables and Views	215
Generating a SELECT Statement	175	Defining Privileges on Packages	216
Using an Alternative to a Scanning Routine	176	Chapter 10. Special Topics	221
Ensuring Data Type Equivalence in a		Contents	221
Dynamically Defined Query	178	Using Datetime Values with Durations	222
Summarizing the Fields of the SQLDA	179	Using Durations	222
Using the SQLN Field	180	Resolving Peculiarities of Date Arithmetic	222
Using the SQLD Field in the SQLDA	180	Using Field Procedures	225
Using the PREPARE Statement	181	Assigning Field Procedures to Columns	227
SQL Functions Not Supported in Dynamic		Understanding Field Procedure Rules	227
Statements	182	Ensuring Data Integrity	231
Chapter 7. Using Extended Dynamic		Ensuring Entity Integrity	231
Statements	183	Using Unique Constraints	232
Contents	183	When Creating a View	232
Using Extended Dynamic Statements to		Ensuring Referential Integrity	232
Maintain Packages	184	Switching Application Servers	244
Illustrating the Use of Extended Dynamic		Identifying Switching Options	245
Statements	187	How to Switch Servers	245
Developing a Query Application	187	Accessing a New Application Server	245
Developing a Language Preprocessor	189	Illustrating Sample Code	246
Grouping Extended Dynamic Statements in		Preprocessing the Program on Multiple	
an LUW	193	Application Servers	247
Using COMMIT WORK and ROLLBACK		Condition Handling with LE/VSE	247
WORK Statements	195	Appendix A. Using SQL in Assembler	
Considering Virtual Storage in a Logical		Language	249
Unit of Work	195	Contents	249
Mapping Extended Dynamic Statements to		Using ARIS6ASD, an Assembler Language	
Static and Dynamic Statements	195	Sample Program	250
SQL Functions Not Supported in Extended		Acquiring the SQLDSECT Area	250
Dynamic Statements	197	Imposing Usage Restrictions on the	
Chapter 8. Maintaining Objects Used by a		SQLDSECT Area	251
Program	199	Rules for Using SQL Statements in	
Contents	199	Assembler Language	253
Managing Dbspaces	200	Identifying Rules for Case	253
Defining Dbspaces	200	Declaring Host Variables	253
Modifying the Size of Dbspaces	204	Embedding SQL Statements	255

Using the INCLUDE Statement	256	Using Host Variables in SQL Statements	299
Using Host Variables in SQL Statements	256	Using Long VARCHAR Host Variables	300
Using DBCS Characters in Assembler Language	256	Using Preprocessor Options	300
Handling SQL Errors	256	Handling SQL Errors	302
Using Dynamic SQL Statements in Assembler Language	257	Using Dynamic SQL Statements in COBOL	304
Defining DB2 Server for VSE Data Types for Assembler Language	258	Defining DB2 Server for VSE Data Types for COBOL	305
Using Reentrant Assembler Language Programs	260	Using Reentrant COBOL Programs	308
Using Stored Procedures	265	Using the DYNAM Compiler Option	308
		Using Stored Procedures	308
Appendix B. Using SQL in C	269	Appendix D. Using SQL in FORTRAN	311
Contents	269	Contents	311
Using ARIS6CD, a C Sample Program	270	Using ARIS6FTD, a FORTRAN Sample Program	312
Rules for Using SQL in C	270	Rules for Using SQL in FORTRAN	312
Placing and Continuing SQL Statements	271	Placing and Continuing SQL Statements	312
Delimiting SQL Statements	271	Placing Data Statements	313
Identifying Rules for Case	271	Using FORTRAN Common Areas	313
Identifying Rules for Character Constants	272	Identifying Rules for Case	313
Using the INCLUDE Statement	272	Declaring Host Variables	313
Using the CONNECT Statement	272	Embedding SQL Statements	315
Using the C Compiler Preprocessor	272	Using Host Variables in SQL Statements	315
Declaring Host Variables	272	Using Variable Length Character Strings	315
Using Host Variables in SQL Statements	278	Using DBCS Characters in FORTRAN	318
Using the Pointer Type Attribute	278	Using the INCLUDE Statement	318
Using Host Variables as Function Parameters	280	Using FORTRAN Variables in SQL: Data Conversion Considerations	318
Using C Variables in SQL: Data Conversion Considerations	280	Handling SQL Errors	318
Using C NUL-Terminated Strings and Truncation	281	Handling Program Interrupts	319
Calculating Dates	281	Using Dynamic SQL Statements in FORTRAN	319
Using Trigraphs	282	Restrictions When Using the FORTRAN Preprocessor	320
Using DBCS Characters in C	282	Defining DB2 Server for VSE Data Types for FORTRAN	321
Considering Preprocessor-Generated Statements	282	Appendix E. Using SQL in PL/I	325
Handling SQL Errors	285	Contents	325
Using Dynamic SQL Statements in C	286	Using ARIS6PLD, a PL/I Sample Program	326
Defining DB2 Server for VSE Data Types for C	287	Rules for Using SQL in PL/I	326
Using Reentrant C Programs	289	Placing and Continuing SQL Statements	326
Using Stored Procedures	289	Delimiting SQL Statements	326
		Using the INCLUDE Statement	327
Appendix C. Using SQL in COBOL	293	Declaring Static External Variables	327
Contents	293	Identifying Rules for Case	327
Using ARIS6CBD, a Sample COBOL Program	294	Declaring Host Variables	327
Rules for Using SQL in COBOL	294	Using Host Variables in SQL Statements	330
Placing and Continuing SQL Statements	294	Using PL/I Variables in SQL: Data Conversion Considerations	330
Delimiting SQL Statements	295	Using DBCS Characters in PL/I	331
Identifying Rules for Case	296	Using SQL Statements in PL/I Subroutines	331
Declaring Host Variables	296	Coding the SIZE Parameter in VSE JCL	332

Handling SQL Errors	332	Contents	339
Handling Program Interrupts	333	How to Use the Decision Tables	340
Using Dynamic SQL Statements in PL/I	334	Decision Tables	341
Defining DB2 Server for VSE Data Types for PL/I	336	Bibliography	349
Using Stored Procedures	337	Index	353
Appendix F. Decision Tables to Grant Privileges on Packages	339		

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 Ltd., Department 071, 1150 Eglinton Avenue 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 contains 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.

Programming Interface Information

This manual documents intended Programming Interfaces that allow the customer to write programs to obtain services of DB2 Server for VSE.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States and/or other countries:

- IBM
- BookManager
- CICS
- CICS/VSE
- DATABASE 2
- DataPropagator
- DB2
- DRDA
- IBMLink
- IMS
- Language Environment
- Library Reader

OS/390
SQL/DS
System/390
VM/ESA
VSE/ESA

| Microsoft, Windows, Windows NT, and the Windows logo are trademarks of
| Microsoft Corporation in the United States and/or other countries.

| Other company, product, and service names may be trademarks or service marks
| of others.

About This Manual

This preface:

- Identifies the book's audience and purpose
- Describes the book's organization
- Lists related publications
- Explains how to read the syntax diagrams
- Presents the conventions for describing MIXED data values.

Audience and Purpose of This Book

This book is for application programmers writing programs in assembler language, C, COBOL,¹ FORTRAN, or PL/I. Throughout the book, the term host languages will often be used to refer to any or all of these particular languages.

This book assumes that you can write programs in one of these host languages for a Virtual Storage Extended/Enterprise Systems Architecture (VSE/ESA) operating system. You may also find it useful to know how to use CICS® and ICCF (or equivalent products) for a VSE/ESA system.

The purpose of the book is to explain how to write application programs that use the Structured Query Language (SQL) to access data stored in DATABASE 2 Server for Virtual Storage Extended/Enterprise Systems Architecture (DB2 Server for VSE) tables. To achieve its purpose, the book:

- Introduces basic concepts
- Provides in-depth discussion of complex areas
- Offers tips of what to do and what not to do
- Focuses more on the Data Manipulation Language of SQL than on the Data Definition Language or the Data Control Language. (The details of the latter two components of SQL are of greater interest to the database administrator than to the application programmer.)
- Describes the host language interfaces and the preprocessor process
- Supplements the material with examples
- Acts as a reference pointer to the appropriate chapters of the *DB2 Server for VSE & VM SQL Reference* manual for details on such technical facts as naming conventions, rules, and syntax.

Organization of This Book

The following information provides a brief description of each chapter and appendix in the book.

This preface identifies the audience, the purpose, and the use of the book.

¹ Throughout this book, COBOL is used to represent either OS/VS COBOL, VS COBOL II, or IBM COBOL for VSE; except where noted otherwise.

Summary of Changes for DB2 Version 6 Release 1 describes the new features of DB2 Server for VSE Version 6 Release 1.

Chapter 1, “Getting Started” on page 1 provides an overview of the application server, the SQL language that accesses the application server, and the host application languages that embed the SQL language.

Chapter 2, “Designing a Program” on page 5 describes the basic framework for designing a DB2 Server for VSE application based on its three main parts: the prolog, body, and epilog.

Chapter 3, “Coding the Body of a Program” on page 21 describes the coding entered in the program body to retrieve and manipulate DB2 Server for VSE data. Data retrieval is described in terms of tables, associated views, and the various means of accessing and selecting table data. Data manipulation focuses on inserting, updating, and deleting data.

Chapter 4, “Preprocessing and Running a Program” on page 105 provides information on the steps you take to preprocess and run an application program. These steps include initial preparation of the system, as well as preprocessing, compiling, link-editing, loading, and running the program.

Chapter 5, “Testing and Debugging” on page 145 shows you how to test a new program, process program errors, and monitor program execution.

Chapter 6, “Using Dynamic Statements” on page 159 describes how to dynamically process SQL statements that are specified at run time.

Chapter 7, “Using Extended Dynamic Statements” on page 183 explains how extended dynamic SQL statements can be used to create and maintain packages of SQL statements. The SQL statements that create and maintain the packages are available only in an application written in the assembler language.

Chapter 8, “Maintaining Objects Used by a Program” on page 199 discusses the management of DB2 Server for VSE objects. First it describes the database space (dbspace); then it discusses the data objects used to manage the data itself, including tables, indexes, synonyms, comments, and labels.

Chapter 9, “Assigning Authority and Privileges” on page 213 explains the techniques used to control user access to, and user manipulation of, the data. A section on user access discusses granting and revoking database authority, while a section on privileges describes assigning of user privileges for tables, views and packages.

Chapter 10, “Special Topics” on page 221 covers various special topics, such as ensuring data integrity, that supplement the material in the preceding chapters.

Appendixes A through E describe information specific to each application host language.

Appendix F contains decision tables used by the system to grant privileges on packages.

The Bibliography lists the full titles and order numbers of related publications. It is followed by the Index.

Related Publications

- *DB2 Server for VSE & VM Overview*
- *DB2 Server for VSE & VM Interactive SQL Guide and Reference*
- *DB2 Server for VSE & VM Database Services Utility*
- *DB2 Server for VSE & VM Quick Reference*
- *DB2 Server for VSE & VM SQL Reference*
- *DB2 Server for VSE Messages and Codes.*

You will need to consult the *DB2 Server for VSE & VM SQL Reference* manual extensively for technical details and the sample tables while working with this book. The sample tables are used for many of the examples in this book.

Syntax Notation Conventions

Throughout this manual, syntax is described using the structure defined below.

- Read the syntax diagrams from left to right and from top to bottom, following the path of the line.

The >>— symbol indicates the beginning of a statement or command.

The —> symbol indicates that the statement syntax is continued on the next line.

The >— symbol indicates that a statement is continued from the previous line.

The —>< symbol indicates the end of a statement.

Diagrams of syntactical units that are not complete statements start with the >— symbol and end with the —> symbol.

- Some SQL statements, Interactive SQL (ISQL) commands, or database services utility (DBS Utility) commands can stand alone. For example:

```
▶▶—SAVE—▶▶
```

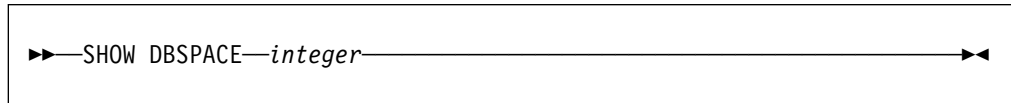
Others must be followed by one or more keywords or variables. For example:

```
▶▶—SET AUTOCOMMIT OFF—▶▶
```

- Keywords may have parameters associated with them which represent user-supplied names or values. These names or values can be specified as either constants or as user-defined variables called *host_variables* (*host_variables* can only be used in programs).

```
▶▶—DROP SYNONYM—synonym—▶▶
```

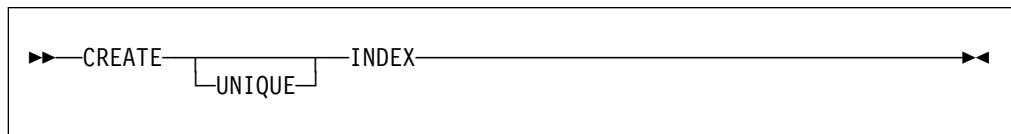
- Keywords appear in either uppercase (for example, SAVE) or mixed case (for example, CHARACTER). All uppercase characters in keywords must be present; you can omit those in lowercase.
- Parameters appear in lowercase and in italics (for example, *synonym*).
- If such symbols as punctuation marks, parentheses, or arithmetic operators are shown, you must use them as indicated by the syntax diagram.
- All items (parameters and keywords) must be separated by one or more blanks.
- Required items appear on the same horizontal line (the main path). For example, the parameter *integer* is a required item in the following command:



This command might appear as:

```
SHOW DBSPACE 1
```

- Optional items appear below the main path. For example:



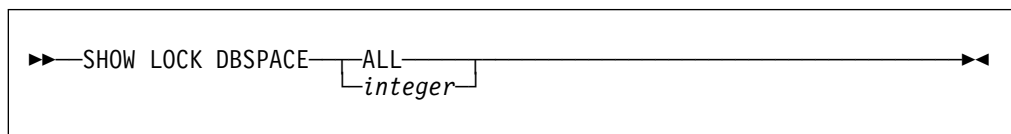
This statement could appear as either:

```
CREATE INDEX
```

or

```
CREATE UNIQUE INDEX
```

- If you can choose from two or more items, they appear vertically in a stack. If you must choose one of the items, one item appears on the main path. For example:



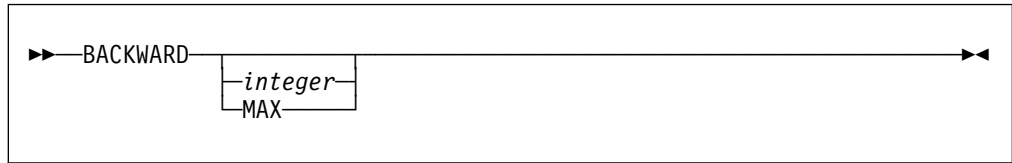
Here, the command could be either:

```
SHOW LOCK DBSPACE ALL
```

or

```
SHOW LOCK DBSPACE 1
```

If choosing one of the items is optional, the entire stack appears below the main path. For example:



Here, the command could be:

BACKWARD

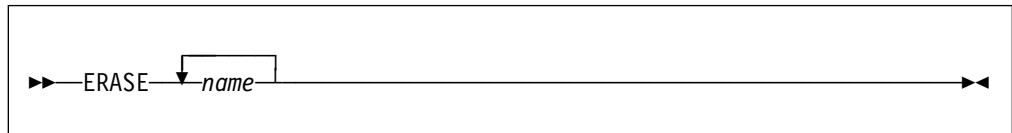
or

BACKWARD 2

or

BACKWARD MAX

- The repeat symbol indicates that an item can be repeated. For example:



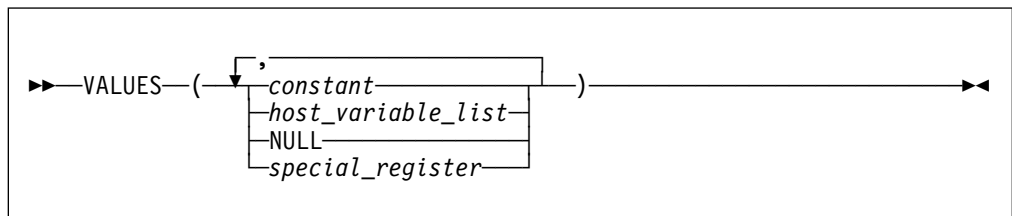
This statement could appear as:

ERASE NAME1

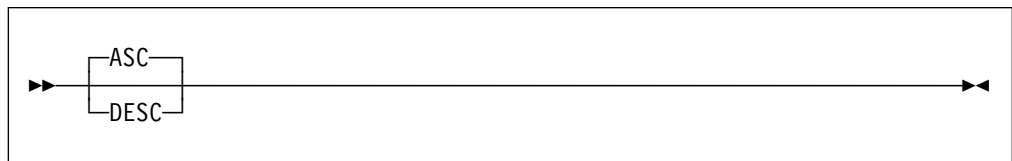
or

ERASE NAME1 NAME2

A repeat symbol above a stack indicates that you can make more than one choice from the stacked items, or repeat a choice. For example:



- If an item is above the main line, it represents a default, which means that it will be used if no other item is specified. In the following example, the ASC keyword appears above the line in a stack with DESC. If neither of these values is specified, the command would be processed with option ASC.

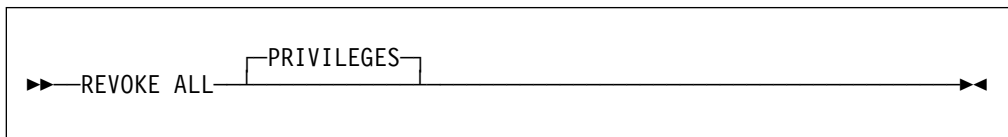


- When an optional keyword is followed on the same path by an optional default parameter, the default parameter is assumed if the keyword is not entered. However, if this keyword is entered, one of its associated optional parameters must also be specified.

In the following example, if you enter the optional keyword PCTFREE =, you also have to specify one of its associated optional parameters. If you do not enter PCTFREE =, the database manager will set it to the default value of 10.

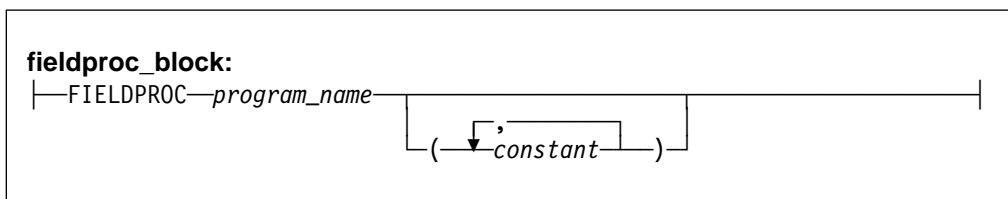
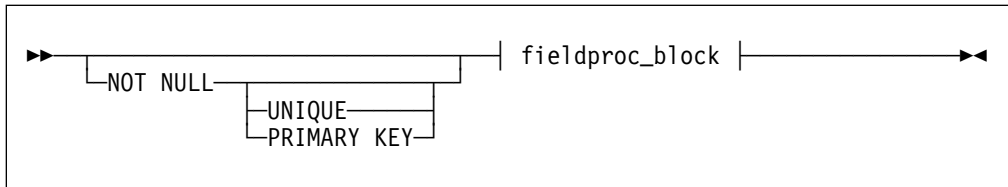


- Words that are only used for readability and have no effect on the execution of the statement are shown as a single uppercase default. For example:



Here, specifying either `REVOKE ALL` or `REVOKE ALL PRIVILEGES` means the same thing.

- Sometimes a single parameter represents a fragment of syntax that is expanded below. In the following example, **fieldproc_block** is such a fragment and it is expanded following the syntax diagram containing it.



SQL Reserved Words

The following words are reserved in the SQL language. They cannot be used in SQL statements except for their defined meaning in the SQL syntax or as host variables, preceded by a colon.

In particular, they cannot be used as names for tables, indexes, columns, views, or dbspaces unless they are enclosed in double quotation marks (").

ACQUIRE	GRANT	RESOURCE
ADD	GRAPHIC	REVOKE
ALL	GROUP	ROLLBACK
ALTER		ROW
AND	HAVING	RUN
ANY		
AS	IDENTIFIED	SCHEDULE
ASC	IN	SELECT
AVG	INDEX	SET
	INSERT	SHARE
BETWEEN	INTO	SOME
BY	IS	STATISTICS
		STORPOOL
CHAR	LIKE	SUM
CHARACTER	LOCK	SYNONYM
COLUMN	LONG	
COMMENT		TABLE
COMMIT	MAX	TO
CONCAT	MIN	
CONNECT	MODE	UNION
COUNT		UNIQUE
CREATE	NAMED	UPDATE
CURRENT	NHEADER	USER
	NOT	
DBA	NULL	VALUES
DBSPACE		VIEW
DELETE	OF	
DESC	ON	WHERE
DISTINCT	OPTION	WITH
DOUBLE	OR	WORK
DROP	ORDER	
EXCLUSIVE	PACKAGE	
EXECUTE	PAGE	
EXISTS	PAGES	
EXPLAIN	PCTFREE	
	PCTINDEX	
FIELDPROC	PRIVATE	
FOR	PRIVILEGES	
FROM	PROGRAM	
	PUBLIC	

Conventions for Representing DBCS Characters

When MIXED data values are shown in examples then the following conventions are used:

Convention	Meaning
<	Represents the DBCS delimiter character X '0E'.
>	Represents the DBCS delimiter character X '0F'.
x	Represents an SBCS character (x can be any lowercase letter).
XX	Represents a DBCS character (XX can be any double-byte uppercase letter).

Components of the Relational Database Management System

Figure 1 depicts a typical configuration with one database, one batch partition user, and a CICS® partition with several interactive users.

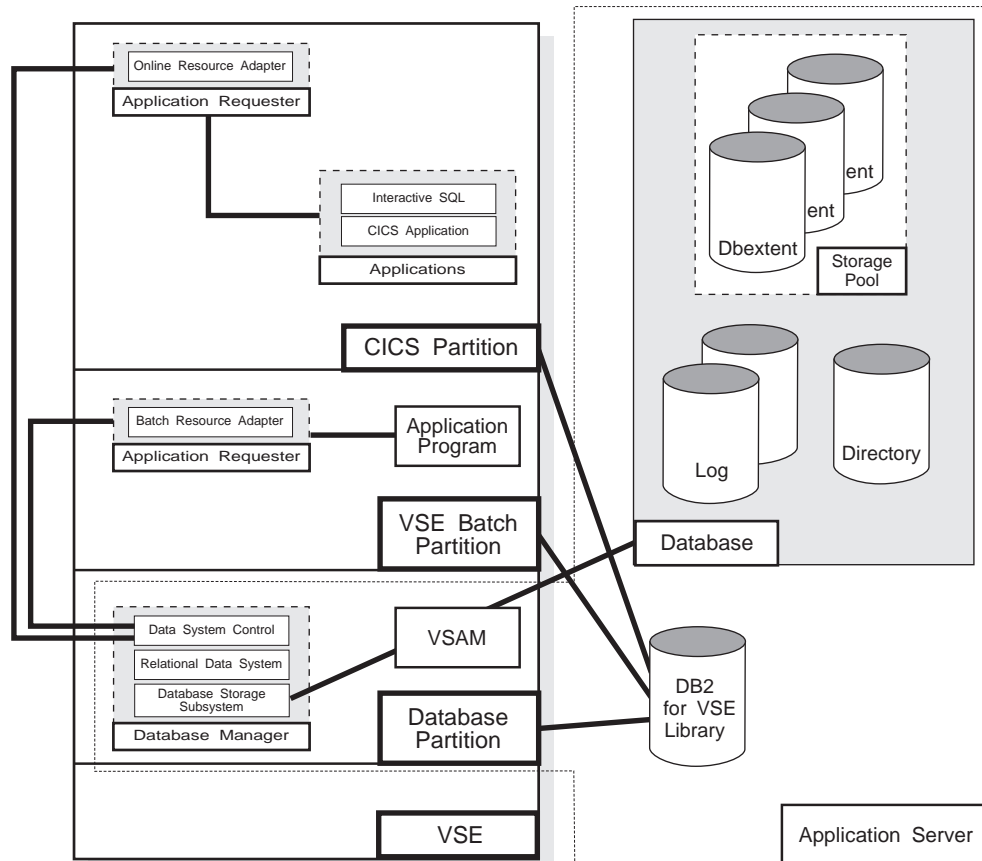


Figure 1. Basic Components of the RDBMS

The **database** is composed of:

- A collection of data contained in one or more *storage pools*, each of which in turn is composed of one or more *database extents (dbextents)*.
- A *directory* that identifies data locations in the storage pools. There is only one directory per database.
- A *log* that contains a record of operations performed on the database. A database can have either one or two logs.

The **database manager** is the program that provides access to the data in the database. It is loaded into the database partition from the DB2 Server for VSE library.

The **application server** is the facility that responds to requests for information from and updates to the database. It is composed of the database and the database manager.

The **application requester** is the facility that transforms a request from an application into a form suitable for communication with an application server.

Summary of Changes for DB2 Version 6 Release 1

This is a summary of the technical changes to the DB2 Server for VSE & VM Version 6 Release 1 database management system. All manuals are affected by some or all of the changes discussed here. This summary does not list incompatibilities between releases of the DB2 Server for VSE & VM product; see either the *DB2 Server for VSE & VM SQL Reference*, *DB2 Server for VM System Administration*, or the *DB2 Server for VSE System Administration* manuals for a discussion of incompatibilities. Version 6 Release 1 of the DB2 Server for VSE & VM database management system is intended to run on the Virtual Machine/Enterprise Systems Architecture (VM/ESA®) Version 2 Release 2 or later environment and on the Virtual Storage Extended/Enterprise Systems Architecture (VSE/ESA™) Version 2 Release 2 or later environment.

Enhancements, New Functions, and New Capabilities

DRDA® RUOW Application Requestor for VSE (Online)

DRDA Remote Unit of Work Application Requestor provides read and update capability in one location in a single unit of work.

This support provides CICS/VSE® online application programs with the ability to execute SQL statements to access and manipulate data managed by any remote application server that implements the DRDA architecture. Online application programs that access remote application servers need to be preprocessed to create a bind file and then bound (using CBND) to the remote application server. Online application programs that access a local application server are preprocessed as in previous releases.

See the following DB2 Server for VSE & VM manuals for further information:

- *DB2 Server for VSE System Administration*
- *DB2 Server for VSE & VM SQL Reference*
- *DB2 Server for VSE Database Administration*
- *DB2 Server for VSE Application Programming*
- *DB2 Server for VSE Installation*

Stored Procedures

The ability to use stored procedures provides distributed solutions that let more people access data faster.

A stored procedure is a user-written application program compiled and stored at the server. When the database is running in multiple user mode, local applications or remote DRDA applications can invoke the stored procedure. SQL statements are local to the server and issued by a stored procedure so they do not incur the high network costs of distributed statements. Instead, a single network send and receive operation is used to invoke a series of SQL statements contained in a stored procedure.

See the following DB2 Server for VSE & VM manuals for further information:

- *DB2 Server for VM System Administration*
- *DB2 Server for VM Database Administration*
- *DB2 Server for VSE & VM SQL Reference*
- *DB2 Server for VSE & VM Operation*

TCP/IP Support for DB2 Server for VM

TCP/IP support allows:

- VM applications to use SQLDS-private protocol to connect to VM databases over TCP/IP.
- VM applications to use DRDA protocol to connect to DB2 family databases (and any other database that supports DRDA connections) over TCP/IP.
- non-VM applications to use DRDA-protocol to access VM database over TCP/IP.

TCP/IP support for DB2 Server for VM integrated with the DB2 Server for VM application server means a system easier to configure and maintain.

The database manager will optionally secure TCP/IP connections using any external security manager that supports the RACROUTE interface.

New Code Page and Euro Symbol Code Page Support

The following CCSIDs are now supported:

- 1112: Latvian/Lithuanian
- 1122: Estonian
- 1123: Ukrainian
- 1130: Vietnamese
- 1132: Lao
- 1148: E-International
- 1140: E-English
- 1141: E-German
- 1144: E-Italian
- 1146: E-UK-English
- 1147: E-French

Additional support has been added for conversions from Unicode (UCS-2) to host CCSIDs.

For a complete list of CCSIDs supported refer to the *DB2 Server for VM System Administration* and *DB2 Server for VSE System Administration* manuals.

DataPropagator™ Capture

DataPropagator Capture is part of the DB2 Family of DataPropagator products. DataPropagator Capture is updated for Version 6 Release 1 compatibility.

QMF for VM, QMF for VSE, and QMF for Windows®

IBM Query Management Facility (QMF™) is now an separately priced feature of DB2 Server for VSE & VM. QMF is a tightly integrated, powerful, and reliable tool that performs query and reporting for IBM's DB2 relational database Management System Family. It offers an easy-to-learn, interactive interface. Users with little or no data processing experience can easily retrieve, create, update, insert, or delete data that is stored in DB2.

QMF offers a total solution that includes accessing large amounts of data and sharing central repositories of queries and enterprise reports. It also allows you to implement tightly-controlled, distributed, or client-server solutions. In addition, you can use QMF to publish reports to the World Wide Web that you can view with your favorite web browser.

Using QMF, users can access a wide variety of data sources, including operational or warehouse data from many platforms: DB2 for VSE, VM, OS/390® and Windows. Via IBM Data Joiner, you can access non-relational data, such as IMS™ and VSAM, as well as data from other vendor platforms.

RDS Above the Line

The RDS component will load and execute above the 16 megabyte line. This support frees up approximately 1.5 megabytes of storage below the 16 megabyte line (or approximately 2.5 megabytes, if DRDA is installed) when compared to Version 5 Release 1. No installation or migration changes are required for this support to be used (except for the definition of VM Shared Segments and for users who execute the database server with AMODE(24)). If sufficient storage is available, the RDS component will be automatically loaded above the 16 megabyte line. When using VM Shared Segments, the RDS Segment should be defined above the 16 megabyte line.

VM users who wish to run the database server in 24-bit addressing mode (i.e. use the AMODE(24) parameter) **must** use a virtual storage size no greater than 16 megabytes. See the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* for release to release incompatibility information.

Combining of NLS Feature Installation Tapes with Base Product Installation Tape

All available NLS features for DB2 Server for VSE, DB2 Server for VM, Control Center for VSE and REXX SQL for VM have been combined with the respective base product installation tape. Customers interested in an NLS feature language will no longer need to order an additional NLS feature tape because all NLS languages will be available to all customers. In all cases, the default language as shipped is American English. The installation and migration processes have been changed to allow you to choose the default language. Refer to the *DB2 Server for VM Program Directory*, *DB2 Server for VSE Installation*, *DB2 for VSE Control Center Installation and Operations Guide*, and *DB2 REXX SQL for VM/ESA Installation* for the details of how these changes affect the installation process and how you can choose to have a different default language.

Control Center Feature

DB2 Server for VSE & VM Version 6 Release 1 enhances the new Control Center feature as follows:

For both VM/ESA and VSE/ESA:

- Access to the Query Management Facility (QMF)

For VM/ESA:

- Compatibility with DB2 Server for VM Version 6 Release 1 initialization parameters and operator commands
- Shared File System Support (SFS) in a VM/ESA environment
- CA-DYNAM/T Interface Support in a VM/ESA environment
- Data Restore Incremental Backup Support in a VM/ESA environment

For VSE/ESA:

- Control Center code installation on any library
- Ability to use while viewing a list of tables online
- Ability to create, reorganize, unload, reload, move and copy tables in batch mode
- Ability to update table statistics in batch mode
- Ability to drop tables online

Data Restore Feature

The Data Restore feature provides archiving and recovery functions in addition to those provided in DB2 for VSE & VM. Data Restore is enhanced in Version 6 Release 1 with incremental database archiving support. The support allows you to archive only the areas of the database that have been updated since the last database archive, instead of having to archive the entire database. This can provide significant savings for customers with large databases which are updated infrequently, or where only a small fraction of the database is updated frequently.

DB2 REXX SQL Feature

The DB2 REXX SQL feature provides a REXX interface for VM customers to allow SQL calls to be executed from REXX programs. The DB2 REXX SQL feature is updated for Version 6 Release 1 compatibility.

Reliability, Availability, and Serviceability Improvements

First failure data capture support is now provided not only on the application server, but also the application requester. This is incorporated with the DRDA RUOW Application Requester support added in this release.

Migration Considerations

Migration is supported from SQL/DS™ Version 3 and DB2 Server for VSE & VM Version 5. Migration from SQL/DS Version 2 Release 2 or earlier releases is not supported. Refer to the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual for migration considerations.

Library Enhancements

Some general library enhancements include:

- The following books have been removed from the library:
 - *DB2 Server for VM Operation*
 - *DB2 Server for VSE Operation*
 - *DB2 Server for VM Interactive SQL Guide and Reference*
 - *DB2 Server for VSE Interactive SQL Guide and Reference*
 - *DB2 Server for VM Database Services Utility*
 - *DB2 Server for VSE Database Services Utility*
- The following books have been added to the library:
 - *DB2 Server for VSE & VM Operation*
 - *DB2 Server for VSE & VM Interactive SQL Guide and Reference*
 - *DB2 Server for VSE & VM Database Services Utility*

Refer to the new *DB2 Server for VSE & VM Overview* for a better understanding of the benefits DB2 Server for VSE & VM can provide.

Chapter 1. Getting Started

Contents

What is the DB2 Server for VSE Product?	2
What is SQL?	3
Embedding SQL Statements in Host Language Programs	3
Writing a Program	4

What is the DB2 Server for VSE Product?

The DB2 Server for VSE product is a database management system that uses the relational data model. You can think of a relational data model as a collection of ordinary two-dimensional tables, where each table has a specific number of columns, unordered rows, and a specific item of data at the intersection of every column and row. You access data by performing operations on tables. All you need to know are the names of tables and of the columns that contain the desired data.

The sample tables in Appendix G of the *DB2 Server for VSE & VM SQL Reference* manual are used in examples throughout this manual. In Figure 2, the DEPARTMENT table has columns DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT.

Figure 2. DEPARTMENT Table Contents

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00
B01	PLANNING	000020	A00
C01	INFORMATION CENTER	000030	A00
D01	DEVELOPMENT CENTER	?	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E01	SUPPORT SERVICES	000050	A00
E11	OPERATIONS	000090	E01
E21	SOFTWARE SUPPORT	000100	E01

Suppose, for example, you want a list of all the different departments (DEPTNAME) in your company. You could get this information simply by knowing the name of the table DEPARTMENT and of the column DEPTNAME that the data is in, and coding this in an appropriate SQL statement.

You can use the DB2 Server for VSE database management system under any supported Virtual Storage Extended (VSE) operating system. Application programs running under VSE can be:

- Online programs operating in CICS partitions, controlled by the Customer Information Control System/Virtual Storage Extended (CICS/VSE)
- Batch programs operating in interactive partitions controlled by the IBM Interactive Communications and Control Facility (ICCF).
- Pure batch programs.

Under the VSE operating system, you can write batch or online programs to access one or more DB2 Server for VSE application servers, or DB2 Server for VM application servers using VSE Guest Sharing. In addition, you can write online programs to access one or more DB2 family application servers using DRDA

Remote Unit of Work (RUOW). The application server is the facility that receives and processes requests to access data.

Access to multiple application servers is not available for CICS application programs; however, CICS programs running in different CICS partitions can access different application servers.

For a discussion of terms and concepts, such as application server, that are used throughout this manual, refer to the *DB2 Server for VSE & VM Overview*, the *DB2 Server for VSE & VM SQL Reference*, and the *DRDA: Every Manager's Guide* manuals.

What is SQL?

DB2 Server for VSE data is handled by the Structured Query Language (SQL), which contains statements that retrieve, delete, insert, and update tables in a DB2 Server for VSE database. You can embed these statements in application programs written in any of the following *host languages*: assembler language, C, COBOL, FORTRAN, or PL/I.

These SQL statements do all data handling, thereby decreasing the data handling done by the programs themselves. Programs that access DB2 Server for VSE data can also access data from other sources, such as DL/I databases.

Embedding SQL Statements in Host Language Programs

Programs that use the DB2 Server for VSE database management system are *host programs* because they act as hosts for SQL. How you embed SQL statements varies for each of the supported host languages.

The core of SQL is the same for each host language. For this reason, the SQL statements are presented throughout this book in *basic form* unless otherwise noted: that is, without any of the language-dependent delimiters.

In this book, examples that have combinations of SQL statements and host language statements are shown in a language-independent form called *pseudocode*. Pseudocode shows program logic but must be recoded in a specific programming language before it can be used. When SQL statements are shown in pseudocode examples, they are preceded by the words EXEC SQL to help you distinguish them from the pseudocode. When shown by themselves, they are not preceded by these words.

To use SQL statements in a programming language, you must be familiar with the rules for embedding them in that language. These rules are discussed in Appendixes A through E (one for each language).

You should browse through the appropriate appendix before you continue reading, and refer to it as needed when you are ready to code your first DB2 Server for VSE application. You can also refer to Chapter 6 of the *DB2 Server for VSE & VM SQL Reference* manual for information on SQL statements.

Writing a Program

Writing a program that accesses DB2 Server for VSE data consists of the following steps: **Designing** the program entails determining what tasks the program must perform, and then creating a plan for the program to perform these tasks. The structure of the program should be based on its three main parts: prolog, body, and epilog. **Coding** the program entails using SQL statements and tools to manipulate DB2 Server for VSE data. The operations on the data must conform to the design of the program. **Preparing** the program for execution entails preprocessing, compiling, link-editing, and loading it. **Testing** and **debugging** the program entails:

- Executing the program using test data
- Checking the results
- Identifying errors created in the previous steps
- Correcting the errors.

Releasing the program entails putting it into production (that is, making it available to its intended users). In this step, you control who will be allowed to run the program and to work with the data that it accesses.

Chapter 2. Designing a Program

Contents

Defining the Main Parts of a Program	6
Creating the Prolog	6
Declaring Variables That Interact with the Database Manager	6
Relating Host Variables to an SQL Statement	6
Handling Errors with the SQL Communications Area	9
Using Additional Nonexecutable Statements	10
Creating the Body	10
Connecting to the Application Server	10
Defining Objects	11
Manipulating Objects	12
Controlling Application Server Resources	12
Granting Authorities and Privileges	12
Creating the Epilog	12
Ending the Program	12
Ending the Program for VSE Batch or ICCF Applications	12
Ending the Program for CICS/VSE Transactions	13
Using Logical Units of Work	14
Defining the Logical Unit of Work	14
Beginning a Logical Unit of Work	14
Considering the CICS/VSE Logical Unit of Work	15
Ending a Logical Unit of Work	15
Using the COMMIT Statement	15
Using the ROLLBACK Statement	16
Summary	16
Using Host-Dependent Sample Applications	17

Defining the Main Parts of a Program

A DB2 Server for VSE application program contains three main parts: the prolog, the body, and the epilog. 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 prolog is at the beginning of every program and must contain:

- SQL statements that provide for error handling by setting up the SQL communications area or by declaring an SQLCODE variable.
- Declarations of all variables that the database manager uses to interact with the host program.

The body contains the SQL statements that will enable you to access and manage data. Among the statements included in this section are:

- The CONNECT statement, which establishes a connection to an application server
- Data manipulation statements (for example, the select-statement)
- Data definition statements (for example, the CREATE statement)
- Data control statements (for example, the GRANT statement).

The epilog is at the end of the application program, and contains SQL statements that:

- Save (commit) or do not use (rollback) changes made to data.
- Release the program's connection to the application server.

Creating the Prolog

Declaring Variables That Interact with the Database Manager

All host program variables that interact with the database manager must be declared in an SQL declare section. A program may contain multiple SQL declare sections. An SQL declare section is a group of host program variable declarations that are preceded by the SQL statement `BEGIN DECLARE SECTION` and followed by the SQL statement `END DECLARE SECTION`. Host program variables declared in an SQL declare section are host variables and can be used in host-variable references in SQL statements.

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 Server for VSE 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 Server for VSE data types, shown in Figure 23 on page 43. Each column of every table is assigned a data type when the table is created.

Relating Host Variables to an SQL Statement: Host variables can be used to receive data from the database manager or to transfer data from the host program to the database manager. Host variables that receive data from the database

manager are **output host variables**. Host variables that transfer data from the host program to the database manager 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 a row that meets the search criteria is found, 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 and each data type can be related to a host language data type. For example, the INTEGER data type is a 31-bit binary integer. This is equivalent to the following data description entries in each of the host languages, respectively:

COBOL:

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

Assembler:

```
variable-name DS F
```

C:

```
long variable-name;
```

FORTRAN

```
INTEGER variable-name
```

PL/I:

```
DCL variable-name BINARY FIXED(31);
```

All the host language equivalents for a particular DB2 Server for VSE data type are listed at the end of each host language appendix.

After you determine which column a host variable interacts with, you need to find out what DB2 Server for VSE data type that column has. Do this by querying the DB2 Server for VSE catalog, which is a set of tables containing information about all tables created in the database. This catalog is described in the *DB2 Server for VSE & VM SQL Reference* manual.

After you have determined the data types, you can refer to the conversion charts at the end of the host language appendixes, and code the appropriate declarations. Figure 3 shows the declarations in each host language.

Figure 3 (Page 1 of 2). Examples of Declarations and Embedded SQL Statements

Assembler	Col. 1	Col. 16	Col. 72
	<pre> EXEC SQL HDATE DS LVL DS IDNO DS EXEC SQL EXEC SQL EXEC SQL EXEC SQL . . . ERRCHK </pre>	<pre> BEGIN DECLARE SECTION CL10 H CL6 END DECLARE SECTION INCLUDE SQLCA WHENEVER SQLERROR GOTO ERRCHK SELECT HIREDATE, EDLEVEL INTO :HDATE, :LVL FROM EMPLOYEE WHERE EMPNO = :IDNO </pre>	<pre> * * * </pre>
C		<pre> EXEC SQL BEGIN DECLARE SECTION; char HDATE[11]; short LVL; char IDNO[7]; EXEC SQL END DECLARE SECTION; EXEC SQL INCLUDE SQLCA; EXEC SQL WHENEVER SQLERROR GOTO ERRCHK; EXEC SQL SELECT HIREDATE, EDLEVEL INTO :HDATE, :LVL FROM EMPLOYEE WHERE EMPNO = :IDNO; . . . ERRCHK: errout(); </pre>	
COBOL	<pre> Cols. 8 12 DATA DIVISION. FILE SECTION. WORKING-STORAGE SECTION. EXEC SQL BEGIN DECLARE SECTION END-EXEC. 01 HDATE PICTURE X(10). 01 LVL PICTURE S9(4) COMPUTATIONAL. 01 IDNO PICTURE X(6). EXEC SQL END DECLARE SECTION END-EXEC. EXEC SQL INCLUDE SQLCA END-EXEC. PROCEDURE DIVISION. EXEC SQL WHENEVER SQLERROR GOTO ERRCHK END-EXEC. EXEC SQL SELECT HIREDATE, EDLEVEL INTO :HDATE, :LVL FROM EMPLOYEE WHERE EMPNO = :IDNO END-EXEC. . . . ERRCHK. </pre>		

Figure 3 (Page 2 of 2). Examples of Declarations and Embedded SQL Statements	
FORTRAN	<pre> Col. 7 EXEC SQL BEGIN DECLARE SECTION CHARACTER*10 HDATE INTEGER*2 LVL CHARACTER*6 IDNO EXEC SQL END DECLARE SECTION EXEC SQL INCLUDE SQLCA EXEC SQL WHENEVER SQLERROR GOTO 4000 EXEC SQL SELECT HIREDATE, EDLEVEL * INTO :HDATE, :LVL * FROM EMPLOYEE * WHERE EMPNO = :IDNO . . . 4000 CONTINUE </pre>
PL/I	<pre> Col. 2 EXEC SQL BEGIN DECLARE SECTION; DCL HDATE CHARACTER(10); DCL LVL BINARY FIXED(15); DCL IDNO CHARACTER(6); EXEC SQL END DECLARE SECTION; EXEC SQL INCLUDE SQLCA; EXEC SQL WHENEVER SQLERROR GOTO ERRCHK; EXEC SQL SELECT HIREDATE, EDLEVEL INTO :HDATE, :LVL FROM EMPLOYEE WHERE EMPNO = :IDNO; . . . ERRCHK: </pre>

Figure 3 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 appendixes of this book.

Handling Errors with the SQL Communications Area

The SQL Communications Area (SQLCA) is discussed in detail in “Using the Automatic Error-Handling Facilities” on page 147. This section presents an overview. To declare the SQLCA, code this statement in your program:

```
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.

The system returns a return code in SQLCODE after executing each SQL statement. The SQLCODE is an integer value that summarizes the execution of the statement. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for a detailed description of the SQLCODE field. Refer to the *DB2 Server for VSE Messages and Codes* manual for information about specific SQLCODEs.

A return code is also returned in SQLSTATE after each SQL statement is executed. SQLSTATE is a character field that provides common error codes across IBM's relational database products. SQLSTATE values comply with the SQL92 standard. For a discussion of SQLSTATE, refer to the *DB2 Server for VSE & VM SQL Reference* manual. For more information about specific SQLSTATES, refer to the *DB2 Server for VSE Messages and Codes* manual.

When a statement is executed successfully, SQLCODE is set to 0 (SQLSTATE is '00000'). A negative SQLCODE indicates an error condition. Positive SQLCODES indicate that a statement has executed successfully but a warning code may be issued which means that you must verify whether the SQL statement was executed without unexpected results.

The system supports the use of a stand-alone SQLCODE. If you request this support, do not include the SQLCA definition in your program. However, you must provide the integer variable SQLCODE (SQLCOD in FORTRAN). For a detailed discussion, see "Using the Automatic Error-Handling Facilities" on page 147.

If you want the system to control error checking after each SQL statement, use the WHENEVER statement. 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 (SQLERROR) occurs, program control is transferred to code that follows a specific 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 *logical unit of work* (LUW) and terminate the program. See "Using Logical Units of Work" on page 14 for more information on LUWs.

Using Additional Nonexecutable Statements

Generally, other *nonexecutable* SQL statements are also part of the prolog. These are discussed later in this manual, and in the *DB2 Server for VSE & VM SQL Reference* manual. Examples of other nonexecutable statements are:

- INCLUDE *text_file_name*
- INCLUDE SQLDA

Creating the Body

Connecting to the Application Server

Your program must establish a connection to the application server before it can run any executable SQL statements. This connection identifies the authorization ID of the user who is running the program, and the name of the application server on which the program will be run.

VSE non-interactive (batch) and ICCF application programs must establish the connection by explicitly issuing the CONNECT statement. You can enter the authorization ID and the name of the target application server. See the *DB2 Server for VSE & VM SQL Reference* manual for a detailed discussion of the CONNECT statement.

The CONNECT statement must be the first SQL statement executed in the batch application. If you release the connection in any logical unit of work other than the last one, issue a new CONNECT statement to reestablish the connection. If the first SQL statement is a CONNECT statement without the TO clause, the default application server is connected. For more information about the defaults that determine which application server is accessed, refer to the *DB2 Server for VSE System Administration* manual. CICS online applications can establish the connection in two ways:

- Issue the CONNECT statement to explicitly request the connection. You can then specify the authorization ID and the name of the target application server. See the *DB2 Server for VSE & VM SQL Reference* manual for a detailed discussion of the CONNECT statement.
- Allow the application to connect implicitly, allowing the user ID and password checking to be performed by the interactive system.

Unless the TO parameter is specified by a CICS/VSE application on a CONNECT statement, the CICS/VSE application will first establish connections to the default application server. On subsequent CONNECTs performed by that application, if the TO parameter is not specified then the connection to the previously connected server will be maintained. For more information about the defaults that determine which application server is accessed, refer to the *DB2 Server for VSE System Administration* manual.

The authorization ID established by the connection must have been granted both the privilege to execute the program's package and CONNECT authority for the target application server. The package has the authority to access database resources specified in the SQL statements in the program if the owner of the package has the authority.

After the connection has been established, your program can issue SQL statements that manipulate data, define and maintain database objects, and begin control operations, such as, granting user authority, and committing changes to the database. See the *DB2 Server for VSE & VM SQL Reference* manual for a more detailed discussion of the CONNECT statement.

Defining Objects

The following are some of the statements that you can use to create and drop database objects such as tables, indexes, and synonyms. (These statements are discussed in Chapter 8, "Maintaining Objects Used by a Program" on page 199.)

- CREATE TABLE
- DROP TABLE
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- CREATE VIEW
- DROP VIEW
- CREATE SYNONYM
- DROP SYNONYM
- CREATE PROCEDURE
- ALTER PROCEDURE
- DROP PROCEDURE
- CREATE PSERVER

- ALTER PSERVER
- DROP PSERVER

Manipulating Objects

The following are some of the statements that you can use to manipulate database objects:

- SELECT
- INSERT
- UPDATE
- DELETE

These statements are discussed in detail in Chapter 3, “Coding the Body of a Program” on page 21 .

Note: Refer to the *DB2 Server for VSE & VM SQL Reference* manual for a description of select-statements.

Controlling Application Server Resources

The following are some of the statements that you can use to manage logical units of work, dbspaces, and locks:

- CONNECT
- ACQUIRE DBSPACE
- DROP DBSPACE
- ALTER DBSPACE
- UPDATE STATISTICS

Granting Authorities and Privileges

There are two statements to use to assign and withdraw privileges on objects or authorities to user IDs:

- GRANT
- REVOKE

They are discussed in detail in Chapter 9, “Assigning Authority and Privileges” on page 213.

Creating the Epilog

Ending the Program

The application epilog is the logical end of your DB2 Server for VSE application program. To properly end your program:

1. End the current logical unit of work (if one is in progress) by explicitly issuing either a COMMIT statement if you want the changes to be committed (saved in the database), or a ROLLBACK statement if you do not want them to be saved.
2. Release your connection to the application server.

The two tasks are accomplished differently for VSE batch or ICCF applications, and for CICS/VSE transactions.

Ending the Program for VSE Batch or ICCF Applications: You can enter either
COMMIT RELEASE

to end the current logical unit of work and commit the changes to the database, or

ROLLBACK RELEASE

to end the current logical unit of work and restore the changes made to the database. The RELEASE keyword is optional; it releases your connection to the application server. You should *always* explicitly end your logical unit of work; however, you should release the connection only when ending the last logical unit of work (if your program has more than one) or when changing your authorization ID or the connected application server. If you release the connection in any logical unit of work other than the last logical unit of work, enter a new CONNECT statement to reestablish the connection. You should not release and reestablish the connection unnecessarily because this may degrade the performance of your program. Begin subsequent logical units of work with an explicit CONNECT statement if the previous logical unit of work was terminated using the RELEASE option.

If you do not code a RELEASE as described above, the system issues one implicitly for you upon task/program termination. Not coding the RELEASE when ending the last logical unit of work is inefficient, however; DB2 Server for VSE resources are held until the application terminates even though you may not be using them.

Note: If you forget to end your logical unit of work, the system interrogates a VSE flag to determine whether the program connection (to the application server) terminated normally or abnormally. If the program terminated normally, the system issues a COMMIT statement on behalf of the program. If the program terminated abnormally, the system issues a ROLLBACK statement.

Once again, to avoid confusion, always explicitly end your logical units of work.

Ending the Program for CICS/VSE Transactions: You can enter

COMMIT

to end the current logical unit of work and commit the changes to the database, or

ROLLBACK

to end the current logical unit of work and restore the changes made to the database. You do not have to explicitly release your connection to the application server (although you can, if you wish). DB2 Server for VSE online support automatically releases the connection for use by other CICS/VSE transactions when the current logical unit of work is committed or rolled back.

If your transaction contains more than one logical unit of work, however, it is not necessary to re-CONNECT to the application server every time you want to start a logical unit of work. When the connection to the database manager is implicitly dropped, DB2 Server for VSE online support remembers the user ID, password, and server-name established in the transaction's original CONNECT. The next time a logical unit of work is begun in that same transaction, online support implicitly issues a CONNECT for you. The re-connection is transparent to the transaction.

You do not have to explicitly issue a COMMIT if that is how you want to end the logical unit of work. A normal transaction termination causes a COMMIT statement to be issued on behalf of the transaction.

A CICS/VSE syncpoint or syncpoint rollback also causes the system to issue a COMMIT or ROLLBACK on behalf of the transaction. Conversely, a DB2 Server for VSE COMMIT or ROLLBACK statement causes a CICS/VSE syncpoint to be taken. If your application is using multiple resources, however, you should issue the SYNCPOINT statement or SYNCPOINT ROLLBACK statement instead of the DB2 Server for VSE COMMIT statement or ROLLBACK statement. Internally, SYNCPOINT statements are always more efficient than the corresponding SQL statements.

Under the CICS/VSE system, an interactive transaction can establish a user exit that will get control at points where an SQL program might be canceled. Control is transferred when the online resource manager is about to wait either for an SQL statement to complete² or for a cross partition link to become available. The user exit can be used to cause the current SQL statement to be canceled. The cancel will cause a -914 SQLCODE (SQLSTATE '57014') to be returned to the transaction and a ROLLBACK to be performed on the logical unit of work. A macro (ARIRCAN) is available to establish the user exit. (The ARIRCAN macro can also be used to set user data for the CIRD transaction.) For more details on the ARIRCAN macro interface and the coding of the exit, see CANCEL Exit in the *DB2 Server for VSE Diagnosis Guide and Reference* manual.

Using Logical Units of Work

Defining the Logical Unit of Work

A logical unit of work (LUW) is a sequence of SQL statements (possibly with intervening host language code) that the database manager treats as a whole.

The system ensures the consistency of data at the LUW level, by ensuring that either *all* operations within an LUW are completed, or *none* are completed. Suppose, for example, that money is to be deducted from one account and added to another. If both these updates are placed in a single LUW, and if a system failure occurs while they are in progress, then when the system is restarted, the data is automatically restored to the state it was in before the LUW began. If a program error occurs, all changes made by the statement in error are restored. Work done in the LUW prior to execution of the statement in error is not undone, unless you specifically roll it back. To determine whether the LUW terminated automatically, you should check the value of SQLWARN6 in the SQLCA. See “Using the Automatic Error-Handling Facilities” on page 147 for more information.

Beginning a Logical Unit of Work

An LUW is begun implicitly with the first *executable* SQL statement and is ended by either a COMMIT or a ROLLBACK statement, or when the program ends.

The following are examples of statements that do not start a logical unit of work:

```
BEGIN DECLARE SECTION          INCLUDE SQLCA
END DECLARE SECTION            INCLUDE SQLDA
WHENEVER
```

² This exit is not available when a transaction is using the DRDA protocol to access remote application servers.

An executable SQL statement always occurs within an LUW. If such a statement is encountered after you end an LUW, it automatically starts another.

Considering the CICS/VSE Logical Unit of Work

For logical unit of work processing to function as described in this manual, **ALL CICS/VSE INSTALLATIONS MUST DO THE FOLLOWING:**

1. The CICS System Initialization Table (DFHSIT) must be generated with DBP=YES.

If this is not done, the CICS/VSE system attempts to commit all changes, regardless of whether a rollback was intended. (Alternatively, DBP=xx can be specified if a suffixed version of the CICS/VSE Dynamic Backout Program is being used.)

2. In addition, each online application that has access to the application server must have Dynamic Transaction Backout set to YES. You can do this by specifying DTB=YES in the assembly of the CICS Program Control Table (DFHPCT).

Your installation can specify DTB=YES on the initial DFHPCT statement, or DTB=YES on each entry DFHPCT statement for applications having access to the database manager.

Note: DTB=NO is not supported in RDO. All transactions defined in the macro with DTB=NO are handled in RDO as if DTB=YES had been specified. For more information, see the *CICS/VSE Resource Definition (Online)* manual.

For more information, refer to the *CICS/VSE System Programming Reference* or the *CICS/VSE Resource Definition (Macro)* manuals.

Ending a Logical Unit of Work

When you end an LUW, you can use either the COMMIT statement to save its changes, or the ROLLBACK statement to ensure that these changes are not saved.

Using the COMMIT Statement

This statement ends the current LUW, and commits any changes made during it.

Changes should be committed as soon as application requirements permit. In particular, programs should be written so that uncommitted changes are not held over a terminal read request, which can result in locks and other resources being held for a long time.

Each application program must explicitly end its LUW before terminating. If you do not end it explicitly, the system automatically commits (upon successful termination of the program) all changes made by the program during its pending LUW unless one of the following conditions occurs:

- A log full condition is encountered.
- Some other system condition occurs that causes database manager processing to end.

See “Creating the Epilog” on page 12 and “Using the Automatic Error-Handling Facilities” on page 147 for more information about program termination.

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

Using the ROLLBACK Statement

This statement ends the current LUW, and restores the data to the state it was in prior to the LUW beginning.

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

Under some circumstances, the system automatically backs out of an LUW. Refer to “Automatically Locking Dbspaces” on page 204 for more information.

Note: 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, 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.

The ROLLBACK statement should not be issued if a severe error occurs (indicated by an S in the SQLWARN0 field of the SQLCA). The only statement that can be issued after a severe error is a CONNECT statement.

Summary

Figure 4 on page 17 summarizes the general framework for a DB2 Server for VSE application in pseudocode format. This framework works for VSE batch or ICCF applications, and for CICS/VSE transactions. This framework must, of course, be tailored to suit your own program.

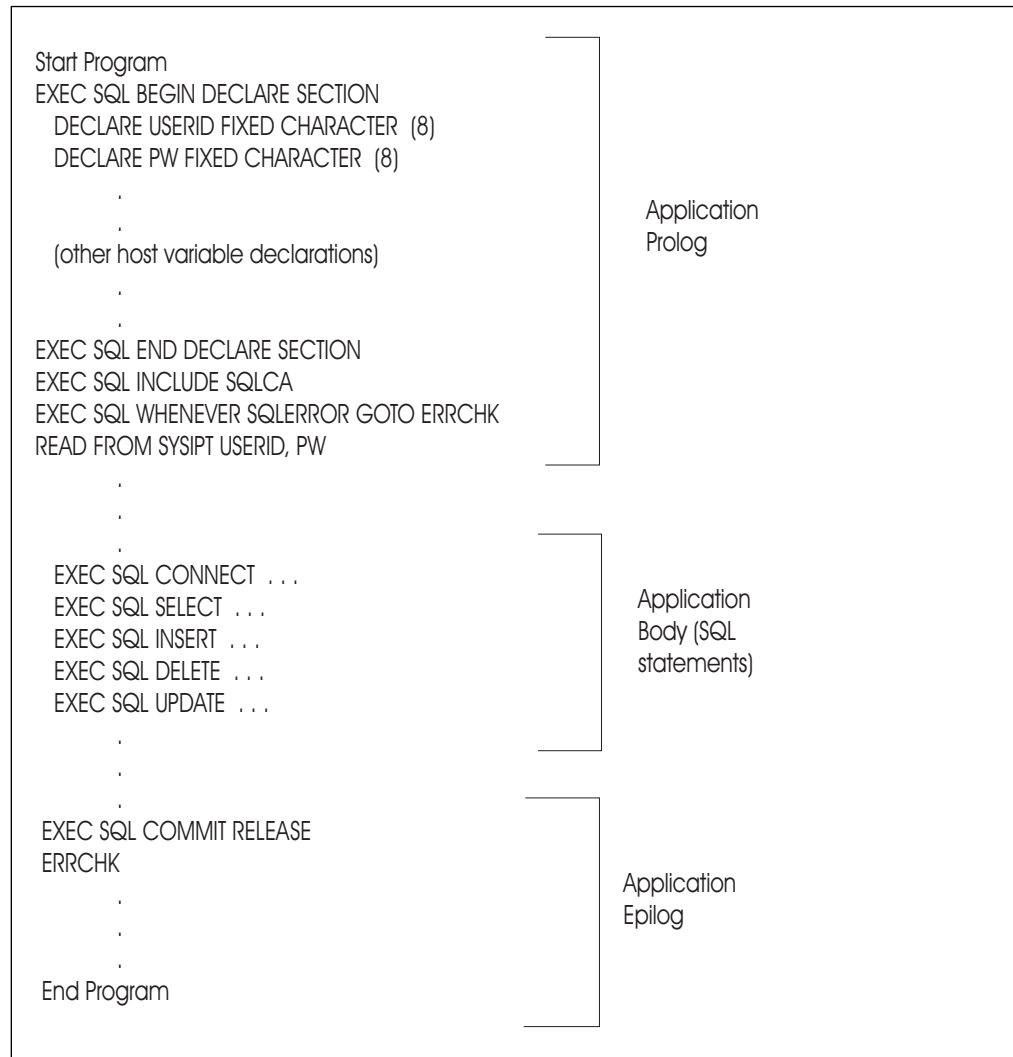


Figure 4. Pseudocode Framework for Coding Programs

Using Host-Dependent Sample Applications

Some host-dependent sample application programs and the JCL streams that can be used to preprocess, compile, link-edit, and run them are shipped with this product. These programs manipulate data in the tables by using embedded SQL statements and printing the results. You may want to model your initial programs from these sample applications. See Figure 5 for information on these samples.

<i>Figure 5. Sample Application Programs</i>			
Language	Program Name	JCL (Z type member)	Appendix
Assembler	ARIS6ASD	ARIS6ASD	A
C	ARIS6CD	ARIS6CD	B
COBOL	ARIS6CBD	ARIS6CBD	C
COBOL II	ARIS6CBD	ARIS6C2D	C
FORTRAN	ARIS6FTD	ARIS6FTD	D
PL/I	ARIS6PLD	ARIS6PLD	E

The sample programs and job control were written for the compiler levels stated in the prolog of the sample programs. If you want to run the sample applications on a different level compiler, refer to the appropriate compiler manual.

Each of the above applications assumes that the user SQLDBA has a password of SQLDBAPW. If the samples are run with a userid other than SQLDBA, or if the password has been changed, the parameters in the generalized JCL must also be changed. Along with these changes, the host variables used by the CONNECT statement in the sample programs must also be modified to reflect a new user ID or password.

```

* *****
* ** GENERALIZED JCL TO PREPROCESS, COMPILE, LINKEDIT AND **
* ** EXECUTE THE SAMPLE PROGRAMS ON VSE SYSTEMS. **
* *****
*
// JOB ARISSAMP PREPROCESS SAMPLE PROGRAM
// EXEC PROC=ARIS61PL *-- DB2 for VSE Library ID PROC
// DLBL SQLGLOB,.....,DISP=(OLD,KEEP) *-- SQLGLOB Parameter file
// ASSGN SYS089,SYSPCH *-- Save SYSPCH assignment
// DLBL IJSYSPH,'PREPROCESSOR.OUTPUT',0 *-- PREPROCESSOR output
// EXTENT SYSPCH,..... *--
ASSGN SYSPCH,... *-- Assign to disk
*
// EXEC PGM=ARIPRPx,SIZE=AUTO,PARM='USERID=SQLDBA/SQLDBAPW, *
PREPNAME=ARIS6xxx' *-- Invoke DB2 for VSE PREPROCESSOR
READ MEMBER ARIS6xxx.A *-- sample program name
/*
CLOSE SYSPCH,SYSPCH *-- Close & Assign SYSPCH
// DLBL IJSYSIN,'PREPROCESSOR.OUTPUT',0 *-- Input File
// EXTENT SYSIPT,..... *-- Same as SYSPCH in
ASSGN SYSIPT,... *-- preprocess step
*
// OPTION CATAL *-- Link Edit (catalog)
PHASE ARIS6xxx,* *-- Name of executable phase
// EXEC compiler *-- Compile
INCLUDE ARIPRDIR *-- DB2 for VSE Batch Resource
Adapter stub
INCLUDE ..... *-- Include runtime routines
*
ENTRY ARIS6xxx *--
// EXEC LNKEDT *-- Link Edit
CLOSE SYSIPT,YSRDR *-- Reset SYSIPT
// ASSGN .... *-- Program assignments
// EXEC PGM=ARIS6xxx,SIZE=(....) *-- Execute Phase
data input to sample program *-- Input data
/*
/&

```

Notes:

1. JCL must be changed to specify the correct device address, DASD extents, compiler references.
2. Replace ARIPRPx with the preprocessor name.
See "Preprocessing the Program" on page 108 for a list of the preprocessor names.
3. Replace ARIS6xxx with sample program name.
4. See "Preprocessing by Mode" on page 109 for a list of preprocessor work files.
5. See "Link-Editing and Loading the Program" on page 132 for a complete list of modules to be included.
6. The SQLGLOB DLBL statement must be added in the JCL if it has not been added to the system standard label subarea.

Figure 6. Generalized Execution JCL for Sample Programs (Multiple User Mode)

Chapter 3. Coding the Body of a Program

Contents

Defining Static SQL Statements	24
Naming Conventions	24
Coding SQL Statements to Retrieve and Manipulate Data	25
Retrieving Data	25
Defining an SQL Query	25
Using the SELECT Clause	27
Using the FROM Clause	29
Using the WHERE Clause	29
Using the GROUP BY Clause	29
Using the HAVING Clause	30
Using the ORDER BY Clause	30
Using the FOR UPDATE OF Clause	31
Using the WITH Clause	32
Retrieving or Inserting Multiple Rows	32
Using the Cursor with a Select-Statement	32
Declaring a Cursor	32
Using a Cursor in an Application Program	33
Manipulating the Cursor	33
The OPEN Statement	33
The FETCH Statement	34
The PUT Statement	35
The Positioned DELETE Statement	35
The Positioned UPDATE Statement	36
The CLOSE Statement	36
Illustrating the Use of the Query Cursor	37
Retrieving Single Rows	37
Constructing Search Conditions	38
Performing Arithmetic Operations	38
Using Null Values	40
Using the Predicates of a Search Condition	40
Evaluating Predicates	41
Using Additional Types of Predicates	42
Using Functions	42
Using Column Functions	42
Using Scalar Functions	43
Using Data Types	43
Assigning Data Types When the Column Is Created	43
Using Long Strings	45
Defining Long Strings	45
Performing Operations on Long Strings	45
Programming Tip	45
Using Datetime Data Types	45
Using Character Subtypes and CCSIDs	46
Determining Default Subtypes and CCSIDs	47
Assigning Subtypes and CCSIDs When a Column Is Created	47
Assigning Subtypes and CCSIDs to Data in a Program	47
Converting Data	48

Summarizing Data Conversion	49
Truncating Data	49
Using a Double-Byte Character Set (DBCS)	50
Using Expressions	51
Using Arithmetic Operators	52
Using Special Registers	53
Concatenating Character and Graphic Strings	54
Using Host Variables	54
Using Host Structures	55
Using Constants	55
Using Numeric Constants	56
Using Character Constants	57
Using Graphic Constants	58
Using Date and Time Constants	58
Using Indicator Variables	59
Notes Common to Both Input and Output Indicator Variables	59
Notes on Input Indicator Variables	59
Notes on Output Indicator Variables	60
Using Views	61
Creating a View	62
Querying Tables through a View	63
Using Views to Manipulate Data	64
Dropping a View	66
Joining Tables	66
Joining Tables Using the Database Manager	66
Performing a Simple Join Query	66
Joining Another User's Tables	67
Analyzing How a Join Works	68
Using VARCHAR and VARGRAPHIC within Join Conditions	68
Using Nulls within Join Conditions	68
Joining a Table to Itself Using a Correlation Name	69
Rules for Table Designation	70
Imposing Limits on Join Queries	71
Using SELECT * In a Join	71
Grouping the Rows of a Table	71
Using VARCHAR and VARGRAPHIC within Groups	72
Using Nulls within Groups	73
Using Select-Lists in Grouped Queries	73
Using a WHERE Clause with a GROUP BY Clause	73
Using the HAVING Clause	73
Combining Joins	74
Illustrating Grouping with an Exercise	74
Nesting Queries	77
Using the IN Predicate with a Subquery	80
Considering Other Subquery Issues	81
Executing Subqueries Repeatedly: Correlation	81
Writing a Correlated Subquery	83
How the Database Manager Does Correlation	83
Illustrating a Correlated Subquery	85
Using a Subquery to Test for the Existence of a Row	90
Table Designation Rule for Correlated Subqueries	90
Combining Queries into a Single Query: UNION	91
String Columns	93
Numeric Columns	93

Datetime/Timestamp Columns	94
SQL Comments within Static SQL Statements	94
Using Stored Procedures	95
Writing Stored Procedures	97
Returning Information from the SQLCA	98
Language Environment® (LE) Considerations	98
Preparing to Run a Stored Procedure	98
Calling Stored Procedures	99
Authorization	100
AUTHIDs	100
Stored Procedure Parameters	100
Datatype Compatibility	101
Conventions for Passing Stored Procedure Parameters	101
The GENERAL Linkage Convention	102
The GENERAL WITH NULLS Linkage Convention	102
Coding Examples	103
Special Considerations for C	103
Special Considerations for PL/I	103
Result Sets	104

Defining Static SQL Statements

This chapter describes how to code SQL statements directly into a program for subsequent preprocessing. These statements which are known in advance of running the program are called *static SQL* statements. Those that are not known until the program is actually run, and have to be built dynamically at run time from input by the user, are called *dynamic* and *extended dynamic SQL* statements. Refer to Chapter 6, “Using Dynamic Statements” on page 159 for a detailed description of dynamic statements and, Chapter 7, “Using Extended Dynamic Statements” on page 183 for a detailed description of extended dynamic statements.

Naming Conventions

The following is a list of the identifiers that must conform in general to specific naming rules:

- Authorization names
- Column names
- Constraint names
- Correlation names
- Cursor names
- Dbspace names
- Descriptor names
- Host variable names
- Index names
- Package names
- Passwords
- Procedure names
- Server names
- Statement names
- Synonyms
- Table names
- View names.

For a description of the naming rules, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

You can access a data object (table, view, dbspace, or package) owned by someone else if you know the owner’s authorization-name and have the appropriate DB2 Server for VSE privileges. You need to qualify references to the object by prefixing its name with the owner’s authorization-name followed by a period. For example, to access the table called EMPLOYEE which is owned by SMITH, enter SMITH.EMPLOYEE.

When you specify the owner along with an object name, you have *fully qualified* the object and *uniquely* identified the table. For example, you cannot have two SMITH.EMPLOYEE tables at the same time.

To avoid confusion and errors, use fully qualified object names. This is especially true if you are coding programs that will be preprocessed by another user.

Coding SQL Statements to Retrieve and Manipulate Data

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

Retrieving Data

One of the most common tasks of an SQL application programmer 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. For example, the following statement will deliver the salary of the employee with the last name of 'HAAS' into the host variable EMPAL:

```
SELECT SALARY
INTO :EMPAL
FROM EMPLOYEE
WHERE LASTNAME='HAAS'
```

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.

Writing select-statements, defining cursors, and using the `SELECT INTO` statement are discussed in the next few sections. For a detailed definition of queries, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

Defining an SQL Query

This section discusses the three forms of a query: the *subselect*, the *fullselect*, and the *select-statement*.

Figure 7 shows the most basic form, the *subselect* query.

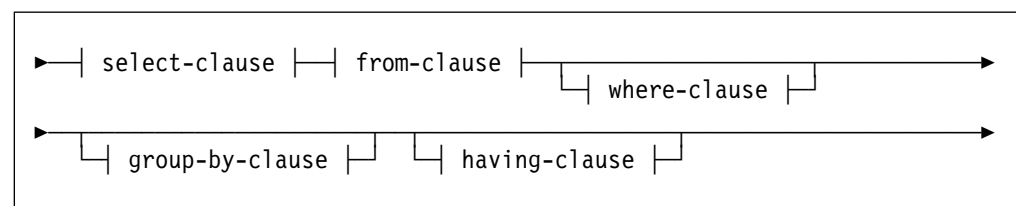


Figure 7. Format of the Subselect

The *subselect* query retrieves the columns specified in the SELECT clause from the tables specified in the FROM clause, applies whatever restrictions the optional clauses; (WHERE, GROUP BY, and HAVING) might put on the scope of the rows selected; and presents the results in a result table, which will be called R. The rows of R are unordered. Only the SELECT clause and the FROM clause are mandatory.

An example of a *subselect* query is:

```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT = 'E11'
```

Figure 8 shows the *fullselect* query.

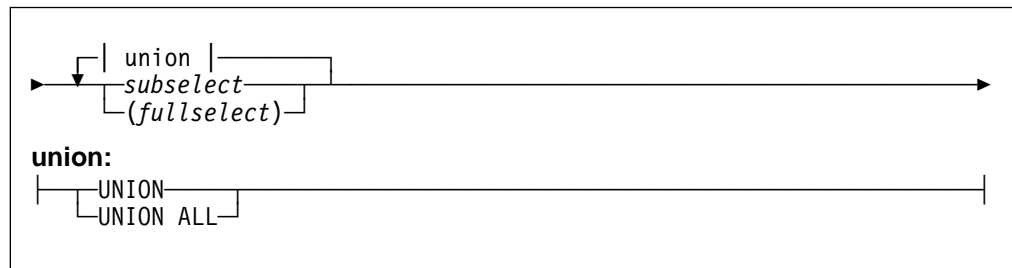


Figure 8. Format of the Fullselect

The *fullselect* query is a merge of two result tables (R1 and R2) from two *subselects* into one final result table (R). The merging is done by the UNION operator. The rows of R are unordered. (For a description of the UNION operation, see “Combining Queries into a Single Query: UNION” on page 91.)

An example of a *fullselect* is:

```
SELECT EMPNO, WORKDEPT, 'EDUCATION'
FROM EMPLOYEE
WHERE EDLEVEL > 16
UNION ALL
SELECT RESPEMP, DEPTNO, 'STAFFING'
FROM PROJECT
WHERE PRSTAFF > 5
```

By using the literal 'EDUCATION' in the first *subselect* and 'STAFFING' in the second, you will be able to tell from R which row was included as a result of which criterion (or query).

Figure 9 shows the *select-statement*.

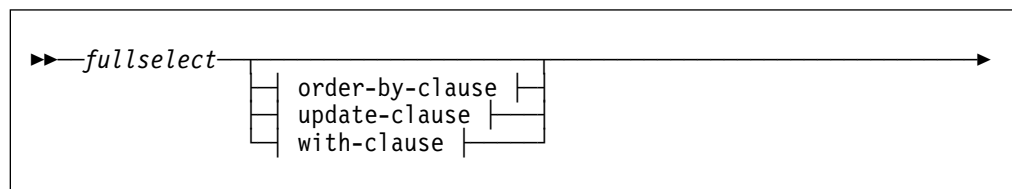


Figure 9. Format of the Select-statement

The *select-statement* can optionally put the rows of R from the *fullselect* in order by the values of the columns identified in the ORDER BY clause. Alternatively, the

select-statement can allow the rows of R to be subsequently updated in the application program, under the restriction that this only be done to those columns listed in the update-clause (FOR UPDATE OF). (This explanation excludes consideration of the preprocessor NOFOR support, which is discussed in the next chapter.) Also, the with-clause may be used to select which isolation level that is to be used by the query. This overrides any other isolation level specification.

An example of a *select-statement* is:

```
SELECT EMPNO, FIRSTNME, LASTNAME, HIREDATE
FROM EMPLOYEE
ORDER BY HIREDATE, LASTNAME
```

Note: In this example, the UNION operator and some of the optional clauses in the *fullselect* are not used.

The distinction among these three forms of query is often quite subtle and academic. It can be useful, however, when other SQL statements specify the form of query that is allowed as part of the statement. For example, CREATE VIEW and INSERT are two statements that use the *subselect*. This tells you that you *cannot* incorporate UNION or ORDER BY in the query component of those statements.

Using the SELECT Clause

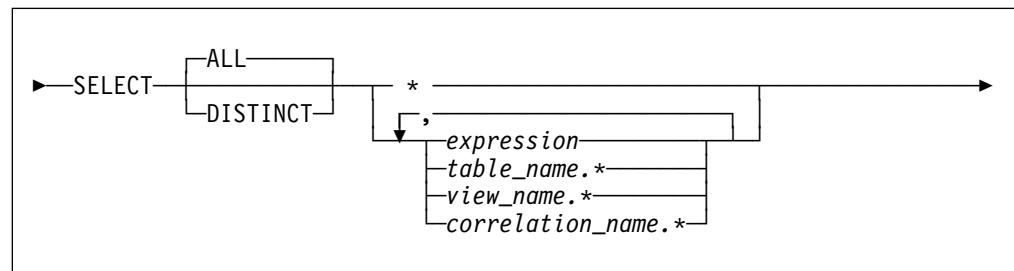


Figure 10. Format of the SELECT clause

This clause is the first part of a *subselect* query. It consists of the keyword SELECT followed by a *select-list*, which usually consists of one or more *expressions*. (Expressions are discussed later in this chapter.)

The following are examples of *select-lists* that can occur in queries to the sample tables:

```
SELECT EMPNO, FIRSTNME, LASTNAME
```

```
SELECT EMPNO, BONUS + COMM
```

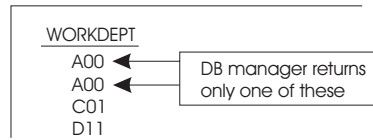
```
SELECT SALARY * 1.10
```

```
SELECT 250
```

```
SELECT HIREDATE + 1 YEAR
```

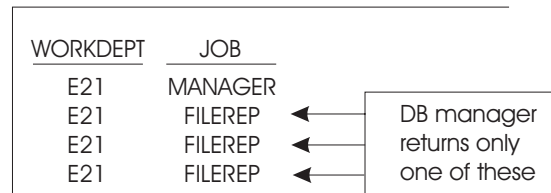
If you specify DISTINCT immediately after SELECT, the system eliminates duplicates from the query-result. (You can use DISTINCT only once in any query.) For example, the following SELECT clause returns the set of different departments:

```
SELECT DISTINCT WORKDEPT
```



Similarly, the following `SELECT` clause returns the set of different departments and jobs:

```
SELECT DISTINCT WORKDEPT, JOB
```



`ALL` indicates that duplicates are not to be eliminated. This is the default.

SQL provides a special shorthand notation for selecting all the columns of a table:

```
SELECT *
```

For example, the following statement returns the entire row from the `DEPARTMENT` table for manager number 000010:

```
SELECT *  
  INTO :DEPART, :NAME, :MGR, :EMPDEPT  
  FROM DEPARTMENT WHERE MGRNO = '000010'
```

As a good programming practice, however, you should explicitly specify every column you want to be returned by your query. This will avoid programming errors when, for example, a new column is added to a table but your program is using `SELECT *` and making no provision to store the extra column value.

If you specify a constant as a *select-list* expression, that constant occurs in every row returned by the query. For example, the following figure shows a query that returns a constant:



An alphabetic constant, such as `'NAME IS'`, is *always* enclosed within single quotation marks (`'`) when used in an SQL statement. A numeric constant should not be enclosed this way.

Using the FROM Clause

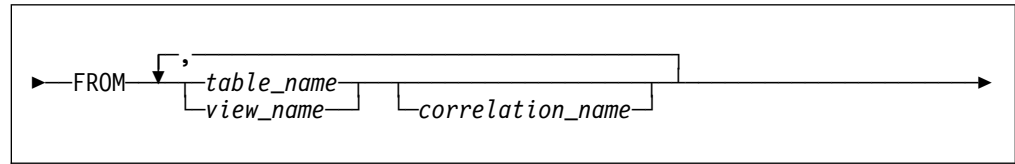


Figure 11. Format of the FROM Clause

This clause specifies the name of the table from which you want to retrieve data. If you are authorized, you can access a table that is owned by someone else, by adding the name of the owner in front of the *table_name* with a period. For example, to specify the table EMPLOYEE owned by user SMITH:

```
FROM SMITH.EMPLOYEE
```

Because any number of users can define a table with the same name, you should always use fully qualified table names. This avoids confusion if you are writing a program that someone else will preprocess.

As Figure 11 indicates, multiple table names are possible, and some or all of these names can have corresponding correlation names. These aspects of the FROM clause are discussed later in this chapter.

Using the WHERE Clause

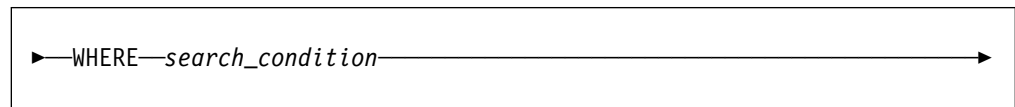


Figure 12. Format of the WHERE Clause

This clause specifies your search conditions. If you do not include it, all the rows of the table will be used to calculate the expressions in the *select-list*. Here are some examples of WHERE clauses:

```
WHERE SALARY > 30000
```

```
WHERE EMPNO = :X
```

```
WHERE SALARY < :R1 AND EDLEVEL = :Y
```

Search conditions are discussed in “Constructing Search Conditions” on page 38.

Using the GROUP BY Clause

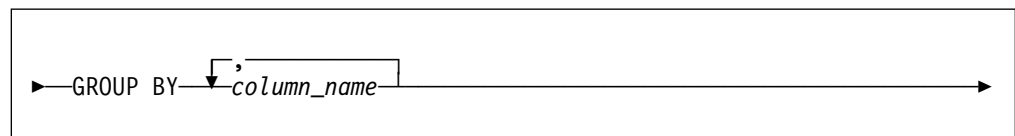


Figure 13. Format of the GROUP BY Clause

This clause enables you to group rows with matching values in one or more columns. Here is an example of the use of the GROUP BY clause:

```

SELECT WORKDEPT, SUM(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT

```

For more information, see “Grouping the Rows of a Table” on page 71.

Using the HAVING Clause

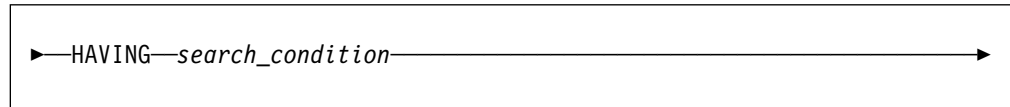


Figure 14. Format of the HAVING Clause

This clause specifies the conditions that must be satisfied by the group. Here is an example:

```

SELECT WORKDEPT, SUM(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT HAVING WORKDEPT <> 'A00'

```

For more information, see “Grouping the Rows of a Table” on page 71.

Using the ORDER BY Clause

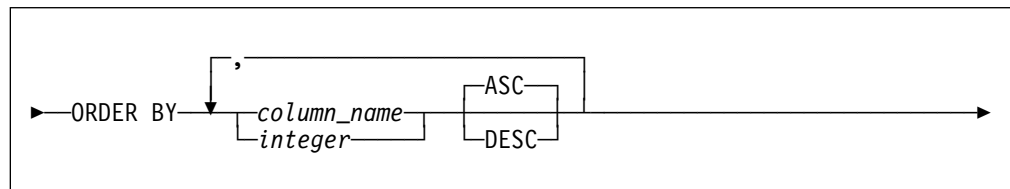


Figure 15. Format of the ORDER BY Clause

This clause delivers the rows of the result table in the order specified. You can indicate order by specifying a list of column names or integers that refer to *select-list* items. For example, ORDER BY 3,5 denotes ordering primarily by the third item and secondarily by the fifth item in the *select-list*. By using integers in the ORDER BY clause, you can order the query result by a selected expression that is not a simple column name.

The following query returns results ordered by the expression SALARY + COMM:

```

SELECT EMPNO, SALARY+COMM
FROM EMPLOYEE
WHERE WORKDEPT='D11'
ORDER BY 2

```

You cannot specify ordering by a column that is not in the *select-list*. For example, the following statement would fail because FIRSTNAME is not in the *select-list*:

```

SELECT SALARY, LASTNAME
FROM EMPLOYEE
ORDER BY FIRSTNAME

```

← Incorrect

The optional word ASC indicates ascending order, and is the default. DESC indicates descending order. ORDER BY 2,5 DESC indicates ascending order on

item 2 and descending order on item 5. Character data is ordered alphabetically, numeric data algebraically, and datetime data chronologically. Null values are sorted first in descending order, and last in ascending order. If you do not specify an ORDER BY clause, rows will be delivered in an order determined by the system.

By default, string data is sorted based on the System/390® collating sequence. However, the collating sequence required for certain alphabets is different from the default System/390 collating sequence. Users expect that sorted data will match the order that is culturally correct for them, and that searches on data will return the result that is correct for the sorting sequence of their language. They are at ease with only one sort order, the one used in their dictionaries, telephone directories, book indexes, and so on.

A way to accommodate special sorting requirements is to use Field Procedures. Field Procedures can be used to encode data being inserted into a column. The encoding effectively alters the collating sequence for the data in the column, enabling the special sorting requirements to be met by the System/390 collating sequence. For more information, see “Using Field Procedures” on page 225.

Trailing blanks in variable string (VARCHAR and VARGRAPHIC) columns do not affect the relative order of rows delivered by the ORDER BY clause. Because the system does not use the trailing blanks when it compares VARCHAR or VARGRAPHIC rows, two columns that differ only by their number of trailing blanks may not maintain their relative positions.

Using the FOR UPDATE OF Clause

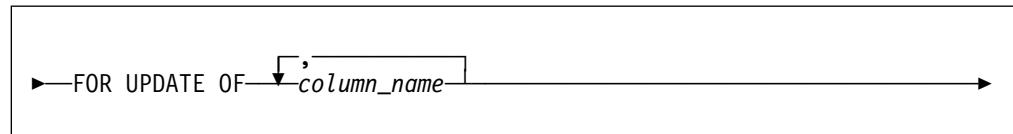


Figure 16. Format of the UPDATE clause

This clause is optional for static SQL if NOFOR support is specified at preprocessor time.

The *update-clause* (FOR UPDATE OF) tells the system that you might want to update some columns of the result table. To update with a cursor, use the WHERE CURRENT OF clause in an UPDATE statement. (See “Manipulating the Cursor” on page 33.) You can update only those columns that you list in the *update-clause*. A column can be in the *update-clause* without being in the *select-list*; therefore, you can update columns that are not explicitly retrieved by the cursor. The *update-clause* is not required for deletion of the current row of a cursor. Deletion with a cursor is done using the WHERE CURRENT OF clause in a DELETE statement. For an explanation of the DELETE statement, see the *DB2 Server for VSE & VM SQL Reference* manual.

Note: If you do not want to be bound by the above restriction on which columns can be updated, you simply invoke NOFOR support at preprocessor time and omit the *update-clause*. In this situation, the preprocessor will assist you by issuing warning or error messages if your program tries to update columns that are not in the current database. If the conditions identified by the warning messages are not corrected, unexpected error messages can subsequently occur at program run time.

Using the WITH Clause

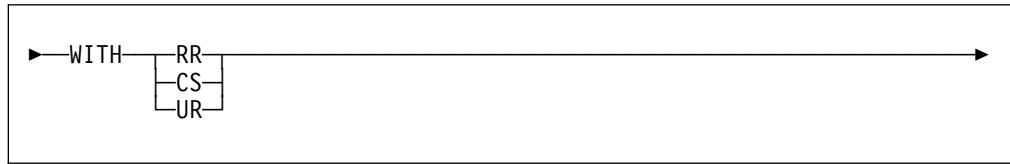


Figure 17. Format of the WITH clause

The WITH clause specifies the isolation level for the query, which overrides any other isolation level specification. For example, a statement specifying WITH UR in a package prepped with ISOL(CS) will use an isolation level of uncommitted read.

For more information on isolation levels, see “Selecting the Isolation Level to Lock Data” on page 124.

Retrieving or Inserting Multiple Rows

Using the Cursor with a Select-Statement

The previous section showed how to use a *select-statement* to create an SQL query. You can now use that query to retrieve values into an application program from multiple rows in a table.

To do so, you must first declare an SQL cursor, which is a control structure that points to a row in a table. The rows returned by the query are called the *result table* of the cursor.

A cursor can be in an *open* or a *closed* state. In the open state, it maintains a position in its result table on a certain row (called the *current row*). If you delete the current row, the cursor will be positioned between the two rows that surrounded the deleted rows. If you request the next row and receive a message that there are no more rows (SQLCODE 100 and SQLSTATE '02000'), the cursor will be positioned after the last row. Before you OPEN the cursor, it is said to be positioned before the first row.

Declaring a Cursor

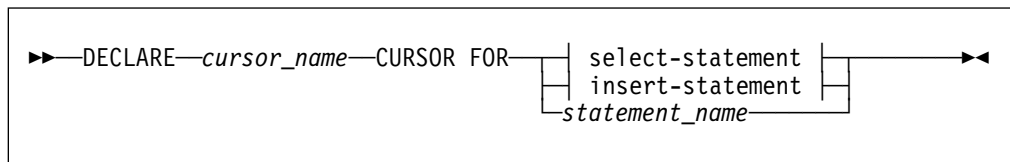


Figure 18. Format of the DECLARE CURSOR statement

Use the DECLARE CURSOR statement to define a cursor. This statement associates a *cursor_name* with a specified *select-statement*, *insert-statement*, or *statement-name*. For example:

```
DECLARE C1 CURSOR FOR SELECT LASTNAME, FIRSTNME
FROM EMPLOYEE WHERE SALARY>:AMT
```

```
DECLARE C2 CURSOR FOR INSERT INTO ACTIVITY
(ACTNO, ACTKWD, ACTDESC)
VALUES (:ACT, :KEYWORD, :DESC)
```

Note: *Statement-name* is only used with dynamic SQL. For an explanation of its use, see “Retrieving the Query Result” on page 171.

The *select-statement* or *insert-statement* is a part of the DECLARE CURSOR statement, so you must not place EXEC SQL in front of SELECT or INSERT (however, do place it in front of the DECLARE).

Using a Cursor in an Application Program

Your program may contain many DECLARE CURSOR statements that define different cursors and associate them with different queries. During the processing of a program, several cursors may be in the open state at one time. It is possible to define more than one cursor that operates on the same data within the same logical unit of work. It is also possible to open a cursor and then operate on the same data with a non-cursor operation such as a Searched DELETE. However, mixing these operations should be avoided, because the result of one operation can adversely affect another. For example, do not update a row using a Positioned UPDATE and subsequently delete it with another cursor operation or with a Searched DELETE.

The DECLARE CURSOR statement that defines a cursor must occur earlier in the program than any statement operating on that cursor. It does not result in any processing when the program is executed (that is, it does not automatically open the cursor).

The *scope* of a cursor-definition is an entire program. Therefore, cursor names must be unique within a program. You cannot have two DECLARE CURSOR statements in the same program that use the same cursor-name, even if they are in different blocks or procedures.

For additional detail on the DECLARE CURSOR statement, see the *DB2 Server for VSE & VM SQL Reference* manual.

Manipulating the Cursor

After you define a cursor, you can manipulate it using the SQL statements shown in Figure 19. (See the *DB2 Server for VSE & VM SQL Reference* manual for a complete description of these statements.)

Figure 19. SQL Statements for Manipulating Cursors

Statements for Manipulating Query and Insert Cursors	Statements for Manipulating Query Cursors	Statements for Manipulating Insert Cursors
OPEN	FETCH	PUT
CLOSE	Positioned DELETE	
	Positioned UPDATE	

The OPEN Statement:

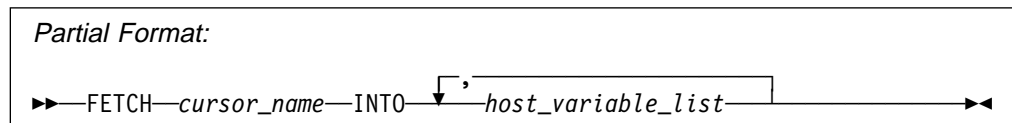
Partial Format:

```
▶—OPEN—cursor_name—▶▶
```

If you are opening a query-cursor (a cursor defined in terms of a select-statement), this statement examines the input host variables (if any) used in the definition of the cursor, determines the result table for the cursor, and leaves it in the open state. When the system executes an OPEN statement for a query-cursor, it positions the cursor *before* the first row of the result table. After the query-cursor is opened, the system does not reexamine its input variables until you close and reopen the cursor. No rows in the result table are fetched to the host program until a FETCH statement is executed. Always open the cursor before issuing the first FETCH or PUT statement.

If you are opening an insert-cursor and your program is blocking, this statement prepares the system to block the rows that are to be inserted. With an insert-cursor, you can change the values of the input host variables between inserts; you do not have to close and reopen the cursor.

The FETCH Statement:



This statement can be executed only when the indicated cursor is in the open state. The position of the cursor is advanced to the next row of the result table, and the selected columns of this row are delivered into the output host variables referenced in the *host_variable_list*.

The following is an example of the FETCH statement:

```

DECLARE QUERY1 CURSOR FOR
SELECT EMPNO, BONUS*1.10
FROM EMPLOYEE
WHERE WORKDEPT='D11'

OPEN QUERY1
FETCH QUERY1 INTO :E1, :B1
  
```

The values are returned in these host variables.

A cursor can move forward only when it is in its result table; the system cannot return to rows that have already been fetched (other than closing the cursor and reopening it).

If the result table of the cursor is empty, or if all its rows have already been fetched, the system returns the *not found* return code (SQLCODE=100 and SQLSTATE='02000') and the cursor is positioned after the last row of the result table. To perform further operations with the cursor, you must close and reopen it.

It is possible for two or more rows in the result table to have exactly the same values. (For example, many rows of the EMPLOYEE table may have the same WORKDEPT, and you might define a cursor that selects only WORKDEPT from the table.) These duplicate values are not eliminated from the result table unless you specify DISTINCT in the SELECT clause of the DECLARE CURSOR statement.

You can use indicator variables in the INTO clause. (For a detailed discussion of indicator variables, see "Using Indicator Variables" on page 59.) Each main

variable in the INTO clause may, at your option, have an associated indicator variable. If a null value is returned, and you haven't provided an indicator variable, a negative SQLCODE is returned to your program and execution of the statement is halted.

The PUT Statement:

Partial Format:

▶▶—PUT—*cursor_name*————▶▶

This statement can be executed only when the indicated cursor is in the open state. The PUT statement inserts one row of data as defined by a cursor. The contents of input host variables referenced in the *host_variable_list* (defined in the VALUES clause of the DECLARE CURSOR statement for insert) are delivered to the database.

For instance, the following statements insert a new row of data into the EMPLOYEE table:

```
DECLARE CC CURSOR FOR
INSERT INTO EMPLOYEE (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL)
VALUES (:EMP, :FIRST, :MID, :LAST, :ED)

OPEN CC
PUT CC
CLOSE CC
```

The values represented by the host variables :EMP, :FIRST, :MID, :LAST, and :ED are placed into the corresponding columns of the new row. The other columns are assigned the null value.

After the PUT statement is executed, you can assign different values to the input host variables to add another row. Alternatively, you can place constants in the VALUES clause of the DECLARE CURSOR statement instead of host variables. This causes identical values to be inserted into the related columns for each PUT.

The PUT statement is used mostly for inserting multiple rows of data into a table in groups or blocks (although, it also works with non-blocked inserts). Blocked inserts are specified with the BLOCK preprocessor parameter. If blocking is in effect, rows are not inserted until the block is full, or until a CLOSE statement is issued. For information on preprocessing your program with the BLOCK option specified, see “Preprocessing the Program” on page 108.

The Positioned DELETE Statement

Partial Format:

▶▶—DELETE FROM—*table_name*—WHERE CURRENT OF—*cursor_name*————▶▶

This statement can be executed only when the indicated cursor is in the open state and positioned on a row of the result table. It deletes that particular row from the table. The cursor itself remains where it was; it is considered to be in the *between* position and, cannot be used for further deletions or updates until it is repositioned by a FETCH statement.

From the example under the FETCH statement, you could delete a row from the EMPLOYEE table after doing a FETCH, by issuing:

```
DELETE FROM EMPLOYEE
WHERE CURRENT OF QUERY1
```

The Positioned UPDATE Statement:

Partial Format:

```
►►—UPDATE—table_name—set_clause—WHERE CURRENT OF—cursor_name—►
```

This statement is similar to the DELETE statement, except that it updates the row of the table on which the cursor is positioned rather than deleting it, leaving the position of the cursor unchanged. When using this statement, you must specify the update-clause in the select-statement.

The following example updates the SALARY column of each fetched row of the EMPLOYEE table:

```
DECLARE QUERY2 CURSOR FOR
SELECT LASTNAME, FIRSTNAME, MIDINIT
FROM EMPLOYEE
WHERE WORKDEPT = 'D21'
FOR UPDATE OF SALARY

OPEN QUERY2

FETCH QUERY2 INTO :LAST, :FIRST, :MID

UPDATE EMPLOYEE
SET SALARY = SALARY + :DELTA
WHERE CURRENT OF QUERY2

CLOSE QUERY2
```

The CLOSE Statement:

Format:

```
►►—CLOSE—cursor_variable—◄◄
```

The indicated cursor leaves the open state, and its result table becomes undefined. No FETCH or PUT statement can be executed on the cursor, and no DELETE or UPDATE statement can refer to its current position until the cursor is reopened by an OPEN statement. The CLOSE statement permits the resources associated with maintaining an open cursor to be released. It should be placed in your program so that it is executed as soon as the program is finished using a cursor.

If your program is blocking, you can close an insert-cursor with an incomplete block to insert the remaining rows.

Always close a cursor before committing changes. If changes are committed before an insert cursor (that is being blocked) is closed, an error occurs.

Illustrating the Use of the Query Cursor

Figure 20, which shows a fragment of pseudocode, illustrates the use of a query cursor C1. It finds the employees of all the rows of the EMPLOYEE table whose department number matches host variable DEPT. The FETCH statements retrieve the selected columns successively into host variables EMP, FNAME, and LNAME. After the results are retrieved, they are displayed on the console.

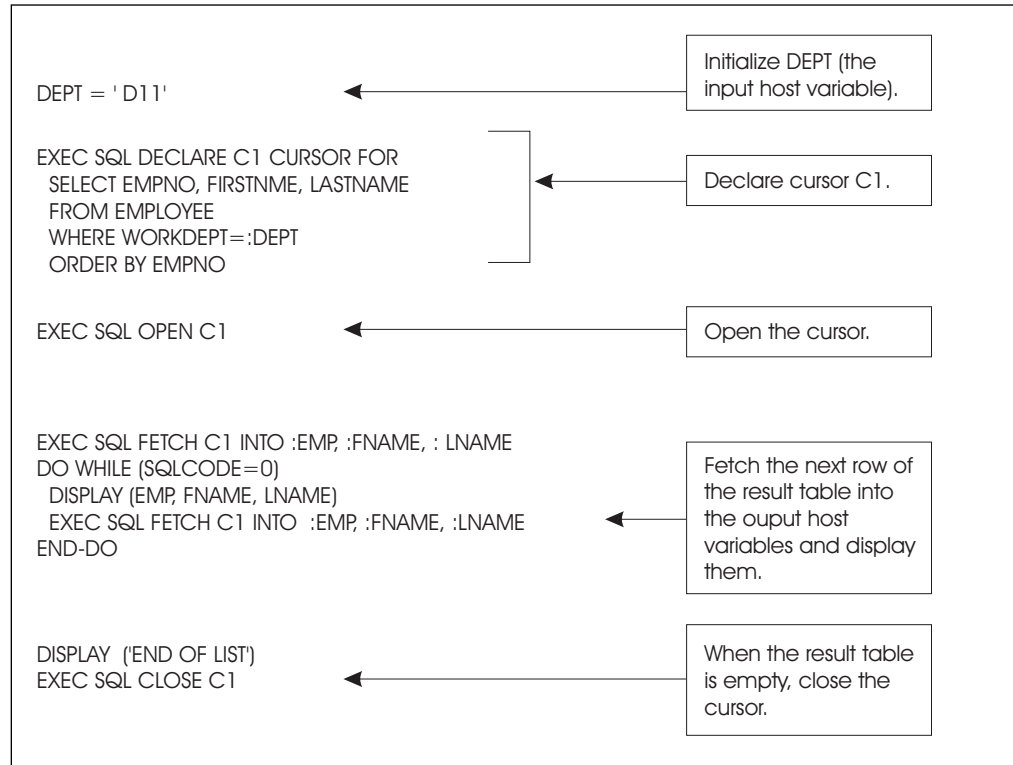


Figure 20. Using a Cursor

Recall that SQLCODE is set to +100 (SQLSTATE '02000') when there are no rows remaining to be fetched.

Retrieving Single Rows

The SELECT INTO statement finds the only row of the table specified in the FROM clause that satisfies the given search condition. From this row, the system selects the columns that you supplied in the *select-list*. The results are inserted in the host variables that you specified in the INTO clause. The data type and length attributes of the host variables must be compatible with the data type and length attributes of the expressions in the select-list. If specified, the WITH clause specifies the isolation level to be used on the query and overrides any other isolation level specification.

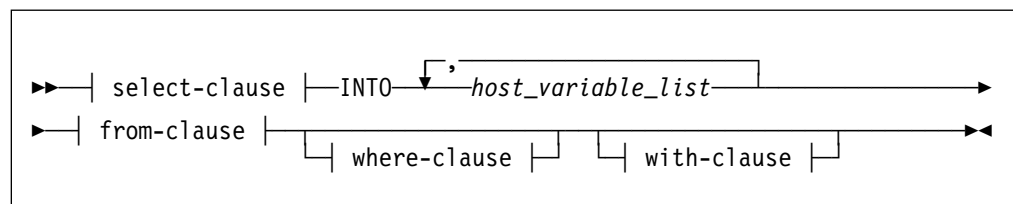


Figure 21. Format of the SELECT INTO statement

For example, the following statement selects the employee number, last name, and yearly salary from the EMPLOYEE table where the employee number is '000130'. It places the result in the host variables EMP, NAME, and PAY:

```
SELECT EMPNO, LASTNAME, SALARY
INTO :EMP, :NAME, :PAY
FROM EMPLOYEE
WHERE EMPNO = '000130'
```

If the number of expressions in the *select-list* is greater than the number of output host variables in the INTO clause, a warning flag (called SQLWARN3) in the SQLCA is set to W. Also, if more than one row satisfies the search condition in a SELECT INTO statement, an error condition occurs, and the values of the host variables are unpredictable.

Constructing Search Conditions

One of the most common operations in SQL is to search through a table, choosing certain rows for processing. A *search condition* is the criterion for choosing rows.

In the following select-statement example, CODE = 'A' AND PART='B' AND TYPE='X' constitute the search condition:

```
SELECT * FROM T1
WHERE CODE = 'A' AND PART='B' AND TYPE='X'
```

When you are constructing search conditions, be careful to perform arithmetical operations only on numeric data types, and to make comparisons only among compatible data types. Graphic data types are compatible only with other graphic data types. If you use a host variable in an expression, its host language data type must be compatible with the rest of the expression.

Performing Arithmetic Operations

Whenever an arithmetic or comparison operator has operands of two different types, the database manager evaluates it in the greater of the two types: FLOAT takes precedence over DECIMAL, which takes precedence over INTEGER, which takes precedence over SMALLINT. For example, if the PRICE column is of type INTEGER and has the value 25, the expression PRICE*.5 will evaluate to 12.5, a decimal value. The predicate PRICE*.5=12 is false, because the decimal value forces the predicate to be evaluated in decimal. (Decimal values are stored in System/390™ packed decimal format.)

The system computes all floating-point values in normalized form, as described in the *ESA/390 Principles of Operation* manual. When a floating-point value is stored in a table, it may not be stored exactly as entered. For example, an SQL INSERT statement could specifically insert the constant 3E0 into a column. Internally, however, the value might actually be stored as 2.9999. Floating-point values may become even more imprecise when arithmetic operations are performed on them. You should use the BETWEEN predicate (described later) when comparing floating-point values.

If the operands of an arithmetic or comparison operator are both single-precision and double-precision floating-point data, the former is converted to the latter before any comparison is made or any arithmetic operation performed. If the equals (=) comparison operator compares these two types of data, the result of the

comparison may not be what you expected. In the following examples, column C1 is defined to contain single-precision floating-point data and column C2 is defined to contain double-precision floating-point data:

```
INSERT INTO T1 (C1, C2) VALUES (10.95, 10.95)
```

```
SELECT * FROM T1  
WHERE C1 = 10.95
```

```
SELECT * FROM T1  
WHERE C2 = 10.95
```

```
SELECT * FROM T1  
WHERE C1 = C2
```

The first and second select statements here will return rows that contain the value 10.95. The third select will not return any rows. This is because the 10.95 cannot be exactly expressed as a floating-point value. The double-precision floating-point representation has more significant bits than the single-precision floating-point representation. When the single-precision floating-point value is converted to double-precision float, X'00's are added to the last four bytes of the double-precision equivalent. The single-precision float data is therefore not equal to the double-precision float data and hence the search condition in the last select above is not satisfied.

|
|
|
|
|
|
|
|
|
|

Decimal numbers have a maximum precision of up to 31 digits. In contrast, a double-precision floating point number preserves up to approximately 17 digits. So when a decimal number with precision greater than 17 is promoted to a floating-point number, digits are lost. Because floating-point numbers can have a larger magnitude than decimal numbers, the float data type is higher than the decimal in the data type promotion scheme. The following example shows how this can cause unexpected results:

```
SELECT * FROM DEPARTMENT WHERE 1E0 + 12345678901234567890.1  
= 12345678901234567890.1;
```

You would expect this statement to return no rows, because adding one to a constant makes it unequal to itself. To execute this statement, the system promotes the two decimal numbers to floating-point values. When this is done, all but the first 17 digits are lost. When '1E0' is added to the first decimal number, > it is not large enough to change the converted decimal value. The end result is that both sides of the expression evaluate as being equal. It is therefore important to be careful when combining floating-point and decimal data types in expressions.

Arithmetic operations between two items of type SMALLINT produce a result of type INTEGER, in order to avoid possible overflow problems (as might easily occur in multiplication). When INTEGER or SMALLINT values are used in a division computation, the result is of type INTEGER, and any remainder is dropped. (See "Converting Data" on page 48 for conversion information.)

Using Null Values

The system allows nulls in values in a table. A null is a nonexistent value; that is, it represents a value that is undefined. You can think of a null value as an empty space, or as a space reserved for later insertion of data.

When null values occur within expressions, the value of the expression is also null. For example, in the following predicate both SALARY and COMM may be a null value:

```
SALARY + COMM < 100
-----
expression1      expression2
```

If either SALARY or COMM is null, expression1 above is null.

Using the Predicates of a Search Condition

A search condition is a collection of one or more *predicates*. Each predicate specifies a test that is applied to the rows of the table. You can connect predicates with the logical operators AND and OR. For example:

```
predicate1 AND predicate2 OR predicate3
```

The keyword NOT can be used to negate a predicate:

```
predicate1 AND NOT predicate2
```

The precedence rule among the keywords is as follows:

1. NOT is applied
2. AND is applied
3. OR is applied.

Use parentheses to override this precedence rule if necessary. For example, the search condition in Figure 22 contains three predicates; it is used to find the rows of the EMPLOYEE table pertaining to an employee from department D11 who also has 17 or 18 years of education.

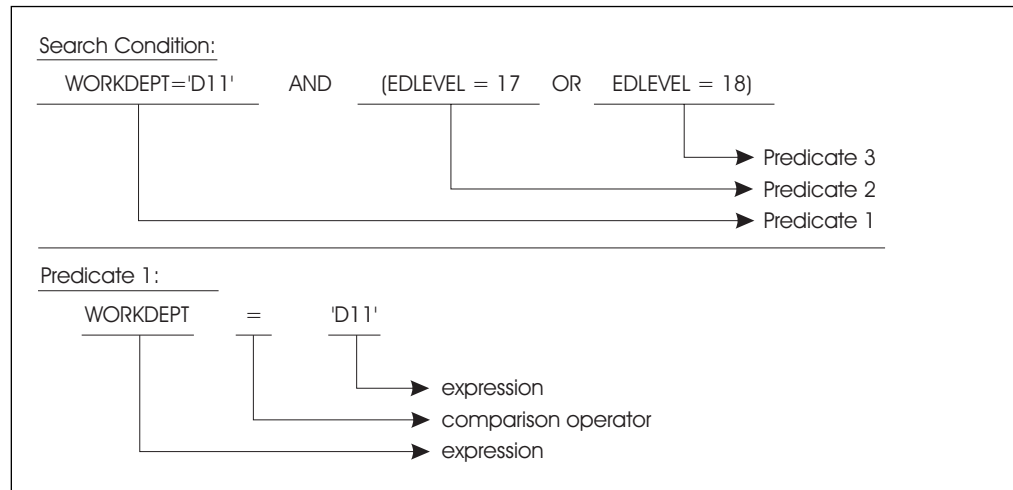


Figure 22. Breakdown of Search Conditions and Predicates

Figure 22 also shows that the format of a predicate is a comparison between two values or expressions. This format is represented as follows:

expression comparison-operator expression

A *comparison-operator* may be any of the following:

- = "equal to"
- ≠ "not equal to"
- <> "not equal to"
- > "greater than"
- >= "greater than or equal to"
- < "less than"
- <= "less than or equal to"

The above symbols are the only comparison operators that you can use in SQL statements. For example, the system does *not* recognize ≠ even if it is supported in the host language. The correct representation of inequality is ≠ or <>.

For a detailed description of search conditions, see the *DB2 Server for VSE & VM SQL Reference* manual.

Evaluating Predicates

The following rules apply when the system evaluates predicates:

1. When two character strings are compared, EBCDIC alphabetic ordering is used. For example:

```
'A' < 'B'
'A' < 'ABLE'
'Z' < '35'
'A1' < 'B'
'a' < 'A'
```

2. When two short strings are compared, trailing blanks are not significant. For example, if the NAME column of a table is of type CHAR(10), you can write NAME='SMITH' in your search condition, and the condition will be satisfied by the database value:

```
'SMITH      '.
```

Trailing blanks are significant in the LIKE predicate; see the *DB2 Server for VSE & VM SQL Reference* manual.

3. In performing an arithmetic operation, if either of the operands is null, the result of the operation is null.
4. In performing a comparison operation, if either of the expressions is null, the result of the comparison is unknown, and the row being evaluated does not qualify for inclusion in the result table.
5. No predicates are permitted on long host variables. Except for LIKE, predicates are not permitted on long columns.
6. When decimal numbers of different scales are compared, the shorter scale is extended with trailing zeros sufficient to match the scale of the larger number. For example, 25.45 is equal to 25.4500.
7. When two graphic strings are compared, the value of the respective data columns is compared in a manner similar to that used for character data types. The single character sequencing is generally of no value for graphic ordering. However, you can specify the sorting sequence of graphic characters in a graphic column by associating the column with a field procedure. For more information on field procedures refer to "Using Field Procedures" on page 225.
8. If a query is executed against an empty table, the database manager may not, for performance reasons, carry out all validation checks. For example, an invalid date string in a host variable is not flagged as an error unless a row is being evaluated.

Using Additional Types of Predicates

In addition to the basic predicates that compare two expressions, the system provides the predicates listed below, which you can use either alone or with other predicates by including the keywords AND, OR, and NOT to form a search condition. For detailed information on the rules and use of these predicates, see the *DB2 Server for VSE & VM SQL Reference* manual.

- BETWEEN
- IN
- LIKE
- NULL
- EXISTS
- Quantified (SOME, ALL).

Using Functions

There are two types of functions. *Column functions* apply the function to a group of values in a column and produce one result value. *Scalar functions* apply the function to one or more values in each row and produce a result value for each row.

Using Column Functions

The column functions are:

AVG MAX MIN SUM COUNT

The argument of a column function is an expression containing a column name (optionally preceded by DISTINCT or ALL— ALL is the default). The argument follows the function and must be enclosed in parentheses.

DISTINCT indicates that duplicate values are to be eliminated before the function is applied. The following example counts the number of different projects that satisfy the search condition:

```
SELECT COUNT(DISTINCT PROJNO)
```

For a detailed discussion of each of the column functions, see the *DB2 Server for VSE & VM SQL Reference* manual.

Using Scalar Functions

The scalar functions are:

CHAR	FLOAT	MINUTE	TIMESTAMP
DATE	HEX	MONTH	TRANSLATE
DAY	HOURL	SECOND	VALUE
DAYS	INTEGER	STRIP	VARGRAPHIC
DECIMAL	LENGTH	SUBSTR	YEAR
DIGITS	MICROSECOND	TIME	

You can use scalar functions wherever an expression can be used. The first or only argument of each scalar function is an expression. If the value of any expression is a null value, the result will be a null value as well, except for the VALUE function.

For a detailed discussion of each of the scalar functions, see the *DB2 Server for VSE & VM SQL Reference* manual.

Using Data Types

Assigning Data Types When the Column Is Created

Each column of every DB2 Server for VSE table is given an SQL data type when the column is created. Figure 23 shows the data types and how they are stored internally.

Figure 23 (Page 1 of 2). SQL Data Types

SQL Data Type	How Stored
INTEGER or INT	Stored as a signed 31-bit binary integer
SMALLINT	Stored as a signed 15-bit binary integer
DECIMAL[(p,s)] or DEC[(p,s)] (4) , (5)	Stored as a packed decimal number of precision <i>p</i> and scale <i>s</i> . Precision is the total number of digits; scale is the number of digits to the right of the decimal point. For example, 251.66 fits in a DECIMAL(5,2) data area. When precision and scale are calculated, if the precision is greater than 31, leading zeros will be removed until it is equal to 31. Trailing zeros are not removed. The default scale is 0 and the default precision is 5.
FLOAT(<i>n</i>) (1)	Stored as a single-precision (4-byte) floating-point number in short System/390 floating-point format, or as a double-precision (8-byte) floating-point number in long System/390 floating-point format.

Figure 23 (Page 2 of 2). SQL Data Types

SQL Data Type	How Stored
CHARACTER[(n)] or CHAR[(n)] (3)	Stored as a character string of fixed length n , where $n \leq 254$. The default length is 1.
VARCHAR(n) (2) , (3)	Stored as a varying-length character string of maximum length n , where $n \leq 32767$. If $254 < n \leq 32767$, VARCHAR(n) is considered a long string.
LONG VARCHAR (3)	Stored as a varying-length character string of maximum length 32767.
GRAPHIC[(n)]	Stored as a string of double-byte character set (DBCS) characters of fixed length n , where $n \leq 127$. The default length is one DBCS character.
VARGRAPHIC(n) (2)	Stored as a varying-length string of DBCS characters of maximum length n , where $n \leq 16383$. If $127 < n \leq 16383$, VARGRAPHIC(n) is considered a long string.
LONG VARGRAPHIC	Stored as a varying-length string of DBCS characters of maximum length 16383.
DATE	Stored as a string of 4 bytes. Each byte is two packed decimal digits. The first two bytes are the year, the next is the month, and the last is the day.
TIME	Stored as a string of 3 bytes. Each byte is two packed decimal digits. The first byte is the hour, the next is the minute, and the last is the second.
TIMESTAMP	Stored as a string of 10 bytes. Each byte is two packed decimal digits. The first 4 bytes are the date, the next 3 are the time, and the last 3 are the microsecond.

Notes:

- The FLOAT data type refers to either single-precision floating-point data (4 bytes) or double-precision floating point-data (8 bytes).
 - REAL and FLOAT(n), where n is from 1 to 21, are synonyms. They are both stored as 4 bytes.
 - FLOAT, DOUBLE PRECISION, and FLOAT(n), where n is from 22 to 53, are synonyms. They are all stored as 8 bytes.
 - When single- and double-precision floating-point data are compared to one another, the result of the comparison may not be what you expected. See “Constructing Search Conditions” on page 38.
- These data types have some special considerations to watch out for.
 - For the CREATE TABLE and ALTER TABLE statements, when VARCHAR(n) or VARGRAPHIC(n) has “ n ” greater than 254 or 127 respectively, the database manager treats the column as a long string when storing and retrieving data. Long strings are discussed in the next section.

The column is treated as VARCHAR or VARGRAPHIC, however, in two respects:

 - The value stored in the LENGTH and SYSLENGTH columns of SYSTEM.SYSCOLUMNS is “ n ”.
 - The value returned to the user in the SQLLEN field of the SQLDA is “ n ”.

When n is less than 255 (VARCHAR) or 128 (VARGRAPHIC) on these statements, the treatment of the column is unchanged.
 - Trailing blanks are not considered relevant in comparisons of VARCHAR or VARGRAPHIC values, unless these values are either concatenated, returned to the application program, or used in a scalar function.

For example, if string X1 = "STRING " and string X2 = "STRING" and X3 = X1 CONCAT X2 then X3 will be equal to "STRING STRING". However, X1 is considered equal to X2 in a compare statement such as a SELECT...WHERE.

3. Columns defined with these data types can contain MIXED or BIT data.
4. NUMERIC is a synonym for DECIMAL, and may be used when creating or altering tables. In such cases, however, the CREATE or ALTER function will establish the column (or columns) as DECIMAL.
5. C application programs can use the decimal data type so that host variables can match table definitions and do not have to do C numeric conversions for table columns that are defined as decimal.

Using Long Strings

Defining Long Strings

A long string column is either a LONG VARCHAR, LONG VARGRAPHIC, VARCHAR(*n*) (where $254 < n \leq 32\,767$), or VARGRAPHIC(*n*) (where $127 < n \leq 16\,383$). Long strings are intended for storage of unstructured data such as text strings, images, and drawings. For a list of restrictions on the use of long strings, refer to the section on data types in the *DB2 Server for VSE & VM SQL Reference* manual.

Performing Operations on Long Strings

The only operations permitted on long strings are:

- SELECT in an outer-level query (not in a subquery).
- INSERT into the database from an input host variable (not from a constant or from a subquery). You can, however, insert null values into long strings with the usual INSERT statement mechanisms. (That is, you are not restricted to host variables when inserting nulls.)
- UPDATE from an input host variable or UPDATE to the null value. (SET LONGFIELD=:X and SET LONGFIELD=NULL are permitted, but SET LONGFIELD='HELLO' and SET LONGFIELD=OTHERFIELD are not permitted.)
- DELETE of rows containing long strings.

Programming Tip

The restrictions on the use of long strings can usually be avoided by the appropriate use of the SUBSTR function.

Using Datetime Data Types

Datetime is a collective DB2 Server for VSE term that includes date, time, and timestamp. Although datetime values can be used in certain arithmetic operations and are compatible with certain strings, they are neither strings nor numbers. Conversely, strings and numbers are not datetime values. A datetime value is either:

- A DATE, TIME, or TIMESTAMP column value
- A value returned by the DATE, TIME, or TIMESTAMP scalar functions
- A value returned by the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers.

Datetime values of the same type can be subtracted. If date1 and date2 are DATE columns, date1 - date2 is a valid expression. Date1 - '01/01/1994' is also a valid expression because '01/01/1994' is a valid string representation of a date. However, '01/01/1994' - '12/20/1932' is not valid because strings cannot be subtracted and a string is interpreted as a date only if the other operand is a value of data type DATE. Scalar functions are provided to explicitly convert strings to datetime values. The following expression is valid:
DATE('01/01/1994') - '12/20/1932'.

For detailed information on the components and valid formats and lengths of the date, time, and timestamp data types and the assignment of these data types to host variables or CHAR-type columns, see the *DB2 Server for VSE & VM SQL Reference* manual.

Using Character Subtypes and CCSIDs

Character subtypes and coded character set identifiers (CCSIDs) provide a means of identifying the character data representation scheme to be used for character and graphic data in your system. For example, by using a certain CCSID, you can specify that all character data in your system is single-byte EBCDIC data.

Subtypes are a way of specifying that you want to use the application server system default CCSID associated with that subtype. CCSIDs apply to both character and graphic data, while subtypes apply only to character data.

For a detailed description of coded character sets and CCSIDs, see the *DB2 Server for VSE & VM SQL Reference* manual.

For most applications, you do not need to specify subtypes or CCSIDs, because the system defaults can usually meet your character data representation requirements.

If this is not the case, you may have to become familiar with Character Data Representation Architecture (CDRA). Refer to the section about data integrity concerns in the *Character Data Representation Architecture Reference and Registry* manual for a discussion of using CDRA to meet your requirements.

The following are examples of problems that can be solved by the specification of CCSIDs or subtypes. The solutions to these problems are discussed in “Assigning Subtypes and CCSIDs When a Column Is Created” on page 47 and “Assigning Subtypes and CCSIDs to Data in a Program” on page 47.

- A column is required in a table to contain mixed data (that is, data that can contain both double-byte and single-byte characters), but the system default specifies that all newly created columns will be used to contain single-byte character set data only.
- A table creation program is required that is to be used at multiple sites, all of which can use different system default subtype and CCSID values. The tables to be created must have the ability to store data of a particular CCSID.
- An application program written in assembler language must insert data into a graphic column, but variables with a graphic data type are not supported.

Determining Default Subtypes and CCSIDs

Refer to the SYSTEM.SYSOPTIONS catalog table to determine the application server system defaults. The rows containing the following values in the SQLOPTION column are important: CHARSUB, CCSIDSBCS, CCSIDMIXED, CCSIDGRAPHIC, and CHARNAME.

Examples of items that assume application requester system defaults are input and output SQLDA elements (the default can be overridden), and host variables.

The following are examples of items that assume application server system defaults:

- Columns (default can be overridden)
- Special registers.

The following are examples of items that assume application requester system defaults:

- Input and output SQLDA elements (default can be overridden)
- Host variables.

For information on setting system defaults, refer to the *DB2 Server for VSE System Administration* manual.

Assigning Subtypes and CCSIDs When a Column Is Created

There are three ways to assign subtypes or CCSIDs to a column:

- Use the application server system defaults.
- Use the preprocessor parameters CHARSUB, CCSIDSBCS, CCSIDMIXED, and CCSIDGRAPHIC to override the system default for columns created by the CREATE TABLE and ALTER TABLE statements in the package. (See “Preprocessing the Program” on page 108 for information on these parameters.)
- Use the subtype or CCSID clause in a column’s definition within the CREATE TABLE or ALTER TABLE statement to override the application server system default or the preprocessor default. (For more information on these statements, refer to the *DB2 Server for VSE & VM SQL Reference* manual.)

Assigning Subtypes and CCSIDs to Data in a Program

There are two ways to assign subtypes or CCSIDs to the data items in a program:

- Use the application server system defaults.
- Execute the SQL statement using dynamic SQL so that the data items can be described in a user-defined SQLDA. A CCSID can be assigned to each data item in the SQLDA.

For examples of how to build an SQLDA that contains CCSID information, see Chapter 6, “Using Dynamic Statements” on page 159. For a more detailed discussion on using the SQLDA, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

Converting Data

For the database manager, the operands in an assignment or comparison operation must be compatible. For example, a character string cannot be compared to a numeric string, a graphic string cannot be compared to a character string, and an arithmetic operation cannot contain a character string operand. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for more details about compatible data types.

Operands that are compatible but are not identical in data types, lengths, datetime formats, or CCSIDs, can be used in assignment and comparison operations but require data conversion as follows:

- For assignment operations, conversion is done before the data is assigned. For example, if a host variable is defined as a SMALLINT field and a column is defined as INTEGER, a SELECT INTO operation converts the INTEGER column to SMALLINT before it is assigned to the host variable. In this situation, overflow may occur if the value is too large to fit into a SMALLINT field. Depending on the data types and the host language, some data may be lost. The *DB2 Server for VSE & VM SQL Reference* manual discusses potential data loss in the assignment of COBOL integers.

To retrieve a datetime value, (that is, a DATE, TIME, or TIMESTAMP), it must be assigned to a character string host variable. The assignment operation converts the datetime value to a character string representation. Whenever a string representation of a datetime value is used in any other operation with a datetime value, the operation is performed with a temporary copy of the string that has been converted to the data type of the datetime value.

If a conversion error occurs when the database manager assigns a value to a host variable in the INTO clause of a SELECT or FETCH statement, and if you have provided an indicator variable for the affected host variable, the system returns the following:

- A value of -2 in the indicator variable
- An undefined value in the host variable
- Warning values in both SQLCODE and SQLSTATE that are appropriate for the condition.

If you have not provided an indicator variable, both SQLCODE and SQLSTATE return error codes (a negative value for SQLCODE, and a data exception for SQLSTATE).

- For comparison operations, one field may be converted if necessary to match the data type, length, or CCSID of another. For example, if two character strings in a comparison operation have different CCSIDs (one is an SBCS string and the other is a mixed string), a temporary copy of the SBCS data is converted to the mixed data CCSID before the data is compared.

For more information about data conversion and conversion errors, see the discussion about assignments and comparisons in the *DB2 Server for VSE & VM SQL Reference* manual.

Summarizing Data Conversion

Data conversion is summarized in tabular form in the *DB2 Server for VSE & VM SQL Reference* manual. Overflow (loss on the left) or truncation (loss on the right) may occur on some conversion attempts.

Truncating Data

Truncations are handled differently for numeric, character, and datetime data.

Numeric data

Truncation of zeros on the left, or of the fractional part of decimal or floating-point values (single-precision or double-precision) takes place without error or warning. Any other loss of data on conversion is an overflow error. If overflow occurs in an outer select and an indicator variable is supplied for the host variable, the indicator variable is set to -2 and a positive SQLCODE is returned; otherwise, a negative SQLCODE is returned.

Character data

When output from the database manager does not fit into a host variable, a warning is returned. SQLWARN1 is set to indicate truncation. In this case, if you provide an indicator variable, the value within it denotes the actual length of the variable in characters before truncation.

When an input character string value does not fit into a DB2 Server for VSE column, an error results.

Whenever truncation occurs, it follows specific rules depending on the character subtype involved. Also, padding may occur when a string is assigned to either a fixed-length host variable or to a fixed-length column and the source string is shorter than the length of the target. Padding, like truncation, follows rules depending on subtype. These rules are in the *DB2 Server for VSE & VM SQL Reference* manual.

SBCS and mixed are the only two types of character data truncation. In mixed truncation, the integrity of target data is ensured. For example, if 'ab< **CCDDEE** >cd' is truncated to a length of 6, the result with mixed truncation is 'ab< **CC** >'. The system counts to byte 6. Because this would split a double-byte character, the number of bytes is rounded to the next lowest whole number. It also always ensures that the < and > characters correctly identify the double-byte characters.

Figure 24 shows the type of truncation that occurs depending on the subtype of the source and target data.

<i>Figure 24. Truncation Types</i>		
Subtype of Source	Subtype of Target	Result
Mixed	Mixed	Mixed truncation
SBCS	SBCS	SBCS truncation
Mixed	SBCS	SBCS truncation ¹
SBCS	Mixed	SBCS truncation ¹
Note:		
1. If the source data contains DBCS data, a conversion error occurs during SBCS truncation.		

Figure 25 shows the results of SBCS and mixed truncation when selecting 'ab< **CCDDEE** >fg' into various host variables:

<i>Figure 25. Examples of Mixed Data Truncation and SBCS Truncation</i>		
Target Host Variable	SBCS Truncation	Mixed Truncation
CHAR(6)	'ab<CCD'	'ab< CC >'
CHAR(7)	'ab<CCDD'	'ab< CC >b'
VARCHAR(7)	'ab<CCDD'	'ab< CC >'

Note: For mixed data, the only difference between the second and the third example is the length of the resulting VARCHAR string. A blank is added to the fixed string.

TIME data

When the seconds part of a retrieved ISO, JIS, or EUR format TIME value is truncated, SQLWARN1 is set to indicate that truncation has occurred. The seconds that are truncated are placed in the indicator variable if one is provided.

TIMESTAMP data

On output, any portion of the microseconds part of a TIMESTAMP may be truncated (including the decimal point). However, no warning is given (SQLWARN1 is not set). If an indicator variable is provided, it is unchanged.

For more information about how computations are performed internally or how overflows can occur, refer to the section about arithmetic operations in the *DB2 Server for VSE Database Administration* manual.

Using a Double-Byte Character Set (DBCS)

DBCS characters can be used in identifiers, constants, and data in DB2 Server for VSE programs. Strings containing DBCS characters are formatted as < **XXXX** >, where < represents the shift-out character, and > represents the shift-in character. Each XX represents one double-byte character set character. The <> delimiters are single-byte character set (SBCS) characters.

In identifiers, characters constants, and character data, the delimiters are significant so redundant delimiter pairs are not removed. For example, the following strings of DBCS characters are not equivalent:

< **AABB** >< **CCDD** > and < **AABBCCDD** >

In graphic data and constants, the delimiters are not significant.

Each DBCS character requires 2 bytes for its representation; therefore, an even number of bytes must be between the < and >. The number of bytes used to represent a string of DBCS characters is equal to:

2 * the number of DBCS characters + 2 (for mixed data)

2 * the number of DBCS characters (for graphic data)

Strings of DBCS characters cannot span lines, whereas mixed strings containing strings of DBCS and SBCS characters can span lines if each string of DBCS characters in the mixed string is on one input record. For a discussion of the rules for using DBCS characters in constants, see “Using Character Constants” on page 57 and “Using Graphic Constants” on page 58.

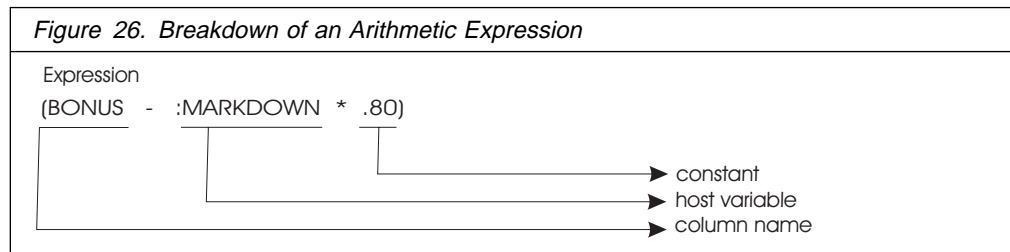
To use DBCS characters in application programs, you must know the following:

- To use host identifiers that contain DBCS characters, the DBCS option in the SYSTEM.SYSOPTIONS catalog must be set to yes. Your compiler must also support DBCS. manual.)
- To use SQL identifiers that contain DBCS characters, the application server must support DBCS characters and mixed data. To verify this for the application server, make sure that in the SYSTEM.SYSOPTIONS catalog table the CHARNAME setting identifies a mixed character set and the DBCS setting is YES. In addition, DBCS characters must be permitted in the particular identifier. For a discussion of rules for using DBCS characters in identifiers, refer to the *DB2 Server for VSE & VM SQL Reference* manual.
- To use host variables with graphic data type, the preprocessor must allow a graphic data type for the host language of the source program. This is true for COBOL and PL/I only. If you need this facility when using another language, see the appendix for that language for a discussion of alternative actions.
- To use graphic and mixed constants (that is, character constants that contain DBCS characters) in an application program, the application server must support mixed data. To verify this for the application server, make sure that the CHARNAME setting in the SYSTEM.SYSOPTIONS catalog table identifies a mixed character set. For a discussion of character sets, refer to the *DB2 Server for VSE System Administration* manual.

Using Expressions

An *expression* refers to a column, a constant, a host variable, an SQL special register (for example, the USER special register), the SQL keyword NULL, a column function, a scalar function, an arithmetic expression, or any of these that can be connected by the concatenation operator. (The concatenation operator is discussed later in this chapter.) Using expressions, you can do calculations on data as part of a query. The calculations are performed before the data is returned to your program.

Figure 26 shows a simple arithmetic expression:



Using Arithmetic Operators

There are four *arithmetic operators* that you can use:

- * multiplication
- / division
- + addition
- subtraction

Usually, the system reads an arithmetic expression from left to right, first applying any negations, then any multiplication or division operations, and then finally any additions and subtractions. For example, in the following expression:

```
BONUS - :MARKDOWN * .80
```

The system would take the value of the host variable `MARKDOWN`, multiply it by `.80`, and then subtract the result from the bonus.

You can change this order-of-precedence by using parentheses. For instance, if the above example were coded:

```
(BONUS - :MARKDOWN) * .80
```

The system would first subtract `MARKDOWN` from `BONUS`, and then multiply the result by `.80`. The two results would probably end up being quite different.

Host variables can be used in arithmetic expressions. For example:

```
PRICE * :QUANTITY + 1.44
```

As mentioned earlier, you must precede the names of host variables by a colon (`:`) to distinguish them from column names. That is, the following is interpreted as a host variable:

```
:PROJNO
```

The following, however, is interpreted as a column name:

```
PROJNO
```

Numeric constants can stand alone or be used in arithmetic combination with other constants or host variables or column names to form expressions. All three of the following are valid expressions:

```
200                      -798.9768                      PRICE * :QUANTITY + 1.44
```

Character constants cannot be used in arithmetic combinations, except when a character string representing a datetime value is used in datetime arithmetic. The following expression is valid:

```
HIRE_DATE - '1994-01-01'
```

The following expression is not valid:

```
'FUDGE'*'GUMDROP'+'LEMON'
```

If you attempt to combine two pieces of data that do not have compatible data types with arithmetic operators, an error code is returned. The system performs data conversion on different types of data that are compatible.

Using Special Registers

Any of the following special registers can be used wherever an expression of the appropriate data type is used:

- CURRENT DATE (defined as DATE)
- CURRENT SERVER (defined as CHAR(18))
- CURRENT TIME (defined as TIME)
- CURRENT TIMESTAMP (defined as TIMESTAMP)
- CURRENT TIMEZONE (defined as DECIMAL(6,0))
- USER (defined as CHAR(8))

Using CURRENT DATE, TIME, and TIMESTAMP: The values of all datetime special registers in the same statement are based on the same time-of-day (TOD) clock reading.

In the examples below, one uses the select-statement and the other uses the UPDATE statement.

```
SELECT CURRENT DATE, PRSTDATE
       FROM PROJECT
       ORDER BY PRSTDATE
```

```
UPDATE PROJECT SET PRSTDATE = CURRENT DATE,
                  PRENDATE = '1996-01-20'
                  WHERE PROJNAME = 'OPERATION'
```

Using CURRENT TIMEZONE: The CURRENT TIMEZONE is a signed time-duration containing the local time zone value. A negative value represents differentials west of the Greenwich-Mean-Time (GMT). A positive value represents differentials east of the GMT. CURRENT TIMEZONE can be used to convert local time into GMT by subtracting CURRENT TIMEZONE from local time. CURRENT TIMEZONE can be subtracted from a TIME or TIMESTAMP data type.

The following example shows a query that involves CURRENT TIMEZONE.

```
SELECT RECEIVED - CURRENT TIMEZONE
       FROM IN_TRAY
```

Using CURRENT SERVER: This special register holds the server name of the application server currently connected. It has a CHAR(18) data type.

The following example shows a query that includes the CURRENT SERVER special register:

```
SELECT ID, INDATE, INTIME
       FROM SAMP1
       WHERE INRDB=CURRENT SERVER
```

Using USER: This special register is evaluated as the currently connected *userid* that is, the user ID of the person who is running the program, regardless of who preprocessed it. USER behaves exactly like a fixed-length character string constant of length 8, with trailing blanks if the user ID has fewer than eight characters.

Notes:

1. You cannot use this keyword in an arithmetic expression (for example, USER+3).
2. You can use it in a predicate where you compare it to a character string (for example, USER = 'JIM').
3. You can use it in the LIKE predicate, where it is treated as a pattern.
4. You can, with some restrictions, use it in the SET clause of an UPDATE statement, or in the VALUES clause of an INSERT statement. In both cases, the data in the target column must be character data type (CHAR or VARCHAR).

The following is a valid expression that includes the USER special register:

```
SELECT *  
FROM SYSTEM.SYSCATALOG  
WHERE CREATOR = USER
```

Concatenating Character and Graphic Strings

You can use the concatenation operator (CONCAT) to concatenate character strings or graphic strings. Long strings cannot be used with the concatenation operator.

The following example shows the concatenation of employees' last names and jobs, separated by a hyphen:

```
SELECT LASTNAME CONCAT '-' CONCAT JOB FROM EMPLOYEE
```

For a full description of this operation, including rules for character subtypes and CCSIDs, see the *DB2 Server for VSE & VM SQL Reference* manual.

Note: The || symbol is a synonym for CONCAT. Because the | symbol is not in a consistent position in all code pages, the use of || could impair code portability.

Using Host Variables

As previously stated, host variables are host program variables that are declared in an SQL declare section. The host program can use these variables to interact with the database manager.

You can use host variables to pass data to or receive data from the database manager. Host variables used to contain column data or data used to evaluate an expression are called main variables. The data type and length attributes of a main variable depend on the data type and length of the column or expression to which the variable relates.

You can also use host variables to communicate information to and from the database manager about the contents of the main variable. If a host variable is used in this context, it is an indicator variable. Only use host variables that are

declared with a data type equivalent to 15-bit integer as indicator variables. Refer to “Using Indicator Variables” on page 59 for a description of their use.

Several SQL statements permit the use of host variables. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for the syntax of these SQL statements. The syntax diagrams indicate whether host variables are permitted or required.

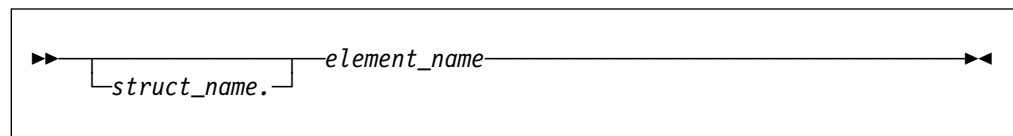
For a description of how to declare host variables, refer to the appropriate host language appendix.

Using Host Structures

A host structure is a special form of host variable. It is any two-level structure or substructure declared in an SQL declare section. Host structures can replace all or part of a *host_variable_list*. A *host_variable_list* can contain references to more than one host structure.

The elements of the host structure comprise the list of main variables in the *host_variable_list*. To provide indicator variable support for the elements of the host structure, you must use an indicator array. An indicator array of n elements provides indicator variable support for the first n elements of the host structure.

The elements of host structures and structures that contain host structures can replace scalar host variables in an SQL statement. You can qualify the element name with the names of parent structures and substructures. The following syntax diagram shows the format of a structure element reference.



It is only necessary to qualify a structure or element name where failure to do so would result in an ambiguous reference.

Elements of indicator arrays cannot be used as host variables and host structures (or structures that contain host structures) cannot be declared as arrays or contain arrays.

Refer to the appropriate host language appendix for rules on the declaration of host structures and indicator arrays. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for more information on the use of host structures and indicator arrays in SQL statements.

Using Constants

Constants (also called *literals*) can be numeric or character data. They are fixed values that can be coded into SQL statements. Like host variables, they are used in various clauses in a number of different SQL statements.

The following example shows a character string constant coded in a WHERE clause:

```

DECLARE C CURSOR FOR
SELECT *
FROM EMPLOYEE
WHERE LASTNAME = 'PEREZ'

```

Constants can be used in the SELECT clause to set up a new column in the result table, which has the specified constant in each of its occurrences. For example, the statement:

```

DECLARE C CURSOR FOR
SELECT LASTNAME, 'WOW', 100.0
FROM EMPLOYEE
WHERE COMM > 3200

```

would have the following result table:

LASTNAME	EXPRESSION 1	EXPRESSION 2
LUCCHESI	WOW	100.0
HAAS	WOW	100.0
THOMPSON	WOW	100.0
GEYER	WOW	100.0

Using Numeric Constants

Integer constants consist of a number with an optional sign, such as -56, 103, or +786. (If you do not include a sign, the system assumes that the number is positive.) All integer constants are 4 bytes long; that is, there are no constants with a data type of SMALLINT.

Decimal constants consist of a number with a decimal point, such as 78.9687, -.00132, 64570., or +1672.80. If you do not supply a decimal point, the constant is interpreted as an integer. In storage, the number occupies a maximum of 16 bytes. Precision p , where $1 \leq p \leq 31$, is the total number of digits. Scale s , where $0 \leq s \leq p$, is the number of those digits that are to the right of the decimal point. Leading and trailing zeros are included in both precision and scale. When the precision and scale are calculated, if the precision is greater than 31, leading zeros are removed until the precision is equal to 31. Trailing zeros are never removed. When decimal data values are multiplied or divided, an overflow condition may occur.

Consider the following:

```
a string of thirty one 9's. * 1.0
```

The string of 9's is treated as DECIMAL(31,0) and 1.0 as DECIMAL(2,1). The precision and scale of the product will then be 31 and 1 (DECIMAL(31,1)), respectively. This will result in a decimal overflow and an arithmetic exception will occur.

This decimal overflow, can be prevented by changing the constant '1.0' to '1.' This would define this constant as DECIMAL (1,0) and the resulting product as DECIMAL (31,0) instead of DECIMAL (31,1). If an expression contains decimal constants, you can influence its precision and scale by adding leading or trailing zeros to those constants.

A floating-point constant is an integer or a decimal constant followed by an exponent marked by the letter E. The E must be followed by an exponent. The 1E0 is acceptable and evaluates to 1. All these are permissible floating-point constants: -2E5, 2.2E-1, .2E6, +5E+2 or 4E0. All floating-point constants are double-precision in the system.

Using Character Constants

Character string constants are coded within quotation marks, and are varying-length character strings of letters, digits, or special characters, such as 'SMITH', '52', or 'k@r -5B'. A character constant implicitly assumes either a FOR SBCS DATA or a FOR MIXED DATA attribute. You cannot assign the FOR BIT DATA attribute to a character constant. The constant is assumed to have a subtype of SBCS unless the following conditions are true. If the following conditions are true, the constant is assigned a subtype of mixed.

- The application server supports mixed data.
- The constant contains mixed data.

Mixed data is composed of a mix of SBCS and DBCS characters in one string. The DBCS portions of the string must be correctly formatted strings of DBCS characters. (For a discussion of the format and rules for using strings of DBCS characters, see "Using a Double-Byte Character Set (DBCS)" on page 50.) An example of mixed data is:

```
'abc< DEFG >hi< JKLM >nop'
```

where abc, hi, and nop represent SBCS characters, and **DEFG** and **JKLM** represent DBCS characters.

To obtain a single quotation mark in a string of SBCS characters, you must code two consecutive single quotation marks. For example, the constant 'DON'T GO' is interpreted as DON'T GO. To obtain a single quotation mark in a string of DBCS characters, you only need to code a single quotation mark. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for more information on mixed strings of SBCS and DBCS characters.

You can also code a character constant using its hexadecimal representation. Hexadecimal constants are treated like regular character constants.

The hexadecimal representation of a constant value must be enclosed within single quotation marks and preceded by an X. For example:

```
X'2D'      X'C1C2C3C4'      X'4256457D'
```

Each pair of hexadecimal numbers (0-9, A-F) represents a single byte. (Either uppercase or lowercase letters can be used.) Therefore, the number of hexadecimal numbers must be even and, when representing a DBCS character in a mixed constant, it must be a multiple of 4 (each DBCS character occupies 2 bytes in storage).

You can use hexadecimal constants to represent SBCS and mixed character data only. The maximum size for hexadecimal constants is 254 hexadecimal digits (that is, 127 SBCS characters or 63 DBCS characters).

The following is a valid expression using a hexadecimal constant:

LASTNAME CONCAT X'FF' CONCAT FIRSTNME

Using Graphic Constants

Graphic string constants are fully supported in COBOL and PL/I programs, but with different formats. The system supports three formats of the graphic constant: the SQL format and two PL/I formats.

The SQL format of the graphic constant is:

```
G'< XXXX >'
```

Note: N is a synonym for G.

The G identifies the constant that follows as graphic; the < XXXX > is any valid string of DBCS characters, and the single quotation marks delimit the constant. You do not need to double the quotation marks in a graphic constant to obtain a single quotation mark. Use this format of the graphic constant in all situations except static SQL statements in PL/I programs.

The PL/I formats of the graphic constant are:

```
1. '< XXXX >'G
```

```
2. < @' XXXX@' @G >
```

Note: N is a synonym for G.

Again, the G indicates that the constant is a graphic constant, and that the string bound by < and > must be a valid string of DBCS characters. In the second format, the single quotation marks and the G are within the string of DBCS characters; they are the DBCS format of the quotation mark and the G. In the second format, to obtain a single DBCS quotation mark, double the occurrence of the DBCS quotation mark within the string of DBCS characters. Use either of these formats of the graphic constant in static SQL statements in PL/I programs.

The PL/I preprocessor converts PL/I format graphic constants into SQL format graphic constants (G'< XXXX >') when they appear in SQL statements. This is done before passing the SQL statement to the application server for processing. Therefore, some DB2 Server for VSE messages for incorrect syntax may refer to the SQL format of the constant, even though a PL/I format constant was coded in your program.

Graphic constants assume the default graphic CCSID. Subtypes do not apply to graphic data. For example, you cannot assign the FOR BIT DATA attribute to a graphic constant. For detailed information on CCSIDs and subtypes, see "Using Character Subtypes and CCSIDs" on page 46.

Using Date and Time Constants

A datetime constant is a character string constant or a decimal constant in a datetime context, as shown in the following examples:

```
END_DATE - '1994-09-13'
```

```
END_DATE - 10000101.
```

In the first example, '1994-09-13' is a datetime character string constant; in the second, 10000101. is a decimal constant. A datetime decimal constant is a date

duration, a time duration, or a timestamp duration. A date duration represents a number of years, months, and days, and is expressed as a DEC(8,0) number. A time duration represents a number of hours, minutes, and seconds, and is expressed as a DEC(6,0) number. A timestamp duration represents a number of years, months, days, hours, minutes, seconds, and microseconds, and is expressed as a DEC (20,6) number.

For more detailed information on date and time values, as well as durations, see “Using Datetime Values with Durations” on page 222.

Using Indicator Variables

Using indicator variables is optional in a host-variable reference. In static SQL statements, indicator variables can be used to indicate that the corresponding host variables should be treated as null values or truncated values. *Output* indicator variables appear in the INTO clause of a SELECT or FETCH statement, and are associated with output that is passed from the database to the application program. *Input* indicator variables appear in the predicates of WHERE and HAVING clauses, in the SET clause of an UPDATE statement, with VALUES in an INSERT statement or in the SELECT clause, and are associated with input that is passed from the application program to the database.

Output indicator variables should always be used wherever null values are allowed in the database. Input indicator variables can be used to put null values into the database. They should, however, not be used in predicates unless there is a very good reason for doing so, because there may be a significant cost in performance.

Refer to the *DB2 Server for VSE & VM SQL Reference* manual for a description of the format of a host-variable reference that contains an indicator variable.

The following example illustrates the use of indicator variables.

```
SELECT FIRSTNAME, LASTNAME
INTO :FNME:FNMEIND, :LNME      :LNMEIND
FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

In this example, the indicator variable FNMEIND provides indicator variable support for the main variable FNME. The indicator variable LNMEIND provides indicator variable support for the main variable LNME.

The following notes on the use of indicator variables are grouped according to the type of indicator variable to which they apply.

Notes Common to Both Input and Output Indicator Variables

1. The indicator variable must be of a host language data type equivalent to an SQL SMALLINT.
2. A negative indicator variable indicates a null value for its main variable.

Notes on Input Indicator Variables

When using input indicator variables, be aware of the following:

1. Input indicator variables can be used to indicate that a column value is to be set to null (when the indicator variable is negative). If you provide an input indicator variable and assign it a negative value, the null value is inserted in the

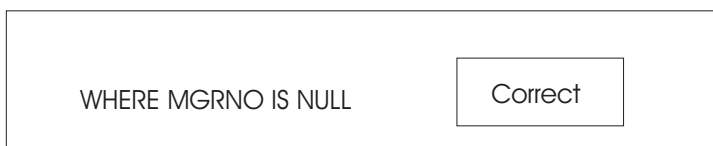
column value for the row. If the indicator variable is zero or a positive value, the main variable is inserted. Truncation does not apply to input variables.

2. A negative indicator variable can be used in static SQL for any of the following predicates:

- The basic comparison ones (such as = or >)
- BETWEEN
- IN
- LIKE
- The quantified ones (ANY, ALL)

See the *DB2 Server for VSE & VM SQL Reference* manual for the different sets of rules for truth values for these predicates.

3. Do not use input indicator variables in search conditions (WHERE or HAVING clauses) to test for null values. The correct way to test for nulls is with the NULL predicate (described earlier):



This will return every row where MGRNO is null.



If MGRIND has been set negative to make MGR null, the truth value is "UNKNOWN", and nothing will be returned.

4. On the other hand, there are cases where setting up a negative input indicator variable in the predicate can prove useful and efficient. For example, if an application prompts the user to interactively supply information that will identify an employee (by either number or name), you can design the program to use only one select-statement to extract the indicated employee data from the database.

Here is the pseudocode:

```
get either empno or lastname from user
if empno is entered then empnoind = 0, else empnoind = -1
if lastname is entered then nameind = 0, else nameind = -1
SELECT * FROM EMPLOYEE
WHERE EMPNO = :EMPNO:EMPNOIND
OR LASTNAME = :NAME:NAMEIND
```

Notes on Output Indicator Variables

When using output indicator variables, be aware of the following:

1. The value returned in an output indicator variable is coded as shown in Figure 27 on page 61.

- Output indicator variables are optional. If a null value is returned, however, and you have not provided an indicator variable, a negative SQLCODE and an error SQLSTATE are returned to your program. If your data is truncated and there is no indicator variable, no error condition results. See “Converting Data” on page 48 for more information about truncation.

Figure 27. Values Returned in Output Indicator Variables

Value Returned	Meaning
0	Denotes that a non-null value that has been returned in the associated host variable is not null.
< 0	Denotes that the value associated with the host variable is null, and should be treated exactly the same way as null column values. A -1 denotes that the null value resulted from a normal operation. A -2 denotes that the null value resulted from either a conversion error or an error while evaluating an arithmetic expression in an outer-select clause.
> 0	Denotes that the system truncated the returned value in the associated host variable because the host variable was not of sufficient length. In addition, if the truncated item was a DBCS character or a string of DBCS characters, the indicator variable contains the length in characters before truncation. If the truncated item was a TIME value, truncated at its seconds part, the indicator variable contains the seconds. The SQLWARN1 warning flag in the SQLCA is set to 'W' whenever truncation occurs.

Using Views

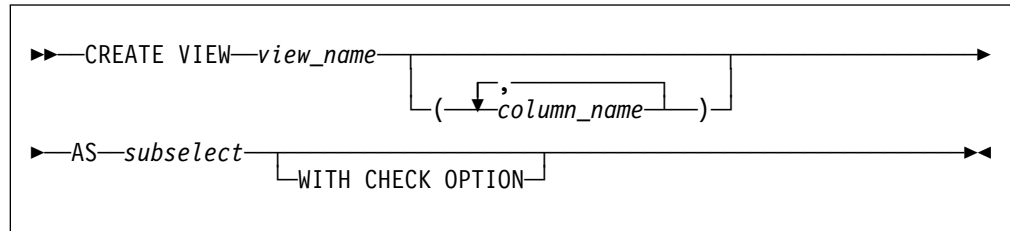
Views allow multiple users to see different presentations of the same data. For example, several users may be operating on a table of data about employees. One may see data about some employees but not others; another may see data about all employees but none of their salaries; and a third may see data about employees joined together with some data from another table. Each of these users is operating on a *view* that is derived from the real table of data about employees. Each view appears to be a table and has a name of its own.

You can create views with authorization statements to control access to sensitive data. For example, you might create a view based on a GROUP BY query that gives certain users access to the *average* salary of employees in each department, but prevents them from seeing any individual salaries.

A view is a dynamic “window” on tables. When you update a real table, you can see the updates through a view; when you update a view, the real table underlying the view is updated. There are, however, restrictions on modifying tables through a view.

Because a view is not physically stored, you cannot create an index on it. However, if you create an index on the real table underlying a view, you may improve the performance of queries on the view.

Creating a View



In the following example, a view is created from the EMPLOYEE table:

```
CREATE VIEW PHONEBOOK (FNAME, LNAME, NUMBER, DEPART, JOBTITLE) AS
  SELECT FIRSTNME, LASTNAME, PHONENO, WORKDEPT, JOB
  FROM EMPLOYEE WHERE JOB <> 'PRES' WITH CHECK OPTION
```

The CREATE VIEW statement causes the indicated *select-statement* to be stored as the definition of a new view, and gives a name to the view and (optionally) to each column in it. If you do not specify the column names, the columns of the view inherit the names of the columns from which they are derived.

You must specify a name for any view column that is not derived directly from a single table column (for example, if a view column is defined as AVG(SALARY) or SALARY+COMMISSION). Columns derived in this manner are often called *virtual columns*, (and contain *virtual data*). You must also specify new column names if the selected columns of the view do not have unique names (for example, if the view is a join of two tables, each of which has a column named PROJNO).

In general, the data types of the columns of the view are inherited from the columns on which they are defined. If a view column is defined on a function, the data type of the view column will be the data type of the function result. (For more details on functions, refer to the *DB2 Server for VSE & VM SQL Reference manual*.)

If you want to prevent the execution of subsequent inserts or updates to the view that involve data that is outside the domain of the view's definition (as specified in the WHERE clause of its subselect), you can add the WITH CHECK OPTION clause. This clause, however, is not allowed for updateable views that are built on subqueries. The checking that is performed at insert or update time is performed according to a set of rules that cover the situation in which a view is dependent on other views. See the *DB2 Server for VSE & VM SQL Reference manual* for these rules.

Some other considerations when creating views are:

- Internal database manager limitations restrict a view to approximately 140 columns. The number of referenced tables, lengths of column names, and WHERE clauses all further reduce this number.
- If the *subselect* in a view definition has a "SELECT *" clause, the view has as many columns as the underlying table. If columns are later added to the underlying table by ALTER statements, the new columns will *not* appear in the view (unless you drop and re-create the view).

- The name of the view must be unique among all the tables, views, and synonyms that you have already created. You can refer to another user's views, if so authorized, by using the owner-name as a prefix (for example, SMITH.PHONEBOOK).
- You can define a view in terms of another view: that is, the *subselect* that defines a view may refer to one or more other views. In this case, follow the rules listed under "Using Views to Manipulate Data" on page 64.
- There is no ORDER BY clause in a *subselect*, therefore, like a table, a view has no intrinsic order. (Of course, you can specify an ORDER BY clause when you write queries against the view.)
- Host variables are not permitted in a CREATE VIEW statement. (For example, predicates such as PRICE = :X are not permitted.)
- The owner of the view is considered to be the authorization ID under which the program is preprocessed.
- When you define a new view, you receive the same privileges that you have on the underlying table. If you possess these privileges with the GRANT option, you can grant privileges on your view to other users. (See Chapter 9, "Assigning Authority and Privileges" on page 213 for information on the GRANT option.) If the view is derived from more than one underlying table, you receive the SELECT privilege, provided that you have this privilege on all the tables from which it is derived. (If you have *no* privileges on the underlying tables, the CREATE VIEW statement returns an error code.) Only the SELECT privilege is possible, because multi-table views do not permit insertion, deletion, or update.
- Primary keys and foreign keys (discussed in "Ensuring Data Integrity" on page 231) cannot be defined on a view.
- If you defined your view on a table that has a primary key, and you make changes to that view, the view should contain all the columns of the key.
- The *subselect* is not executed when the view is created, which means that semantic errors (for example, specifying "WHERE COL = '10'" when COL is a decimal column) are not detected until the view is used. To determine whether a statement contains semantic errors, you can enter 'SELECT *' against the view after creating it.

Querying Tables through a View

You can write queries (select-statements) against views exactly as if they were real tables. When you make a query against a view, the query is combined with the definition of the view to produce a new query against real stored tables. This query is then processed in the usual way. For example, the following query might be written against the view PHONEBOOK that was defined under "Creating a View" on page 62:

```
SELECT FNAME, LNAME
FROM PHONEBOOK
WHERE DEPART = 'D11'
ORDER BY 2
```

The system combines the query with the definition of PHONEBOOK, and processes the resulting internal query:

```

SELECT FIRSTNME, LASTNAME
FROM EMPLOYEE
WHERE JOB <> 'PRES'
AND WORKDEPT = 'D11'
ORDER BY 2

```

During the processing of a query on a view, the system may detect and report errors (by a negative SQLCODE) in either of two phases:

- The combination of the query with the view-definition (for example, attempting to add together two strings of character-type)
- The execution of the resulting query on real tables (for example, attempting to fetch a null value when no indicator variable is provided).

Note: If a view materialization is required to process the view, this view must not contain any LONG VARCHAR columns in the view definition. For a detailed description of view materialization, refer to the *DB2 Server for VSE Database Administration* manual.

Using Views to Manipulate Data

Like select-statements, INSERT, DELETE, and UPDATE statements can be applied to a view just as though it were a real stored table. The SQL statement that operates on the view is combined with the definition of the view to form a new SQL statement that operates on a stored table. Any data modification made by such a statement is visible to users of the view, the underlying table, or other views defined on the same table (if the views “overlap” in the modified area).

The following is an example of an update applied to the view PHONEBOOK, showing how the update can be modified to operate on the real table EMPLOYEE:

View Definition for PHONEBOOK:

```

CREATE VIEW PHONEBOOK (FNAME, LNAME, NUMBER, DEPART, JOBTITLE) AS
SELECT FIRSTNME, LASTNAME, PHONENO, WORKDEPT, JOB
FROM EMPLOYEE WHERE JOB <> 'PRES' WITH CHECK OPTION

```

```

UPDATE PHONEBOOK
SET NUMBER = '9111'
WHERE LNAME = 'SMITH'
AND FNAME = 'DANIEL'

```

becomes:

```

UPDATE EMPLOYEE
SET PHONENO = '9111'
WHERE LASTNAME = 'SMITH'
AND FIRSTNME = 'DANIEL'
AND JOB <> 'PRES'

```

Note: Because of the WITH CHECK OPTION, the following update will not be allowed when Sally takes over as president:

```

UPDATE PHONEBOOK
SET JOBTITLE = 'PRES'
WHERE LNAME = 'KWAN'
AND FNAME = 'SALLY'

```

You must observe the following rules when modifying tables through a view:

1. INSERT, DELETE, and UPDATE of the view are not permitted if the view involves any of the following operations: join, GROUP BY, DISTINCT, or any column function such as AVG.
2. A column of a view can be updated only if it is derived directly from a column of a single stored table. Columns defined by expressions such as SALARY + BONUS or SALARY * 1.25 cannot be updated. (These columns are sometimes called virtual columns.) If a view is defined containing one or more such columns, the owner does not receive the UPDATE privilege on these columns. INSERT statements are not permitted on views containing such columns, but DELETE statements are.
3. The ALTER TABLE, CREATE INDEX, and UPDATE STATISTICS statements cannot be applied to a view.

You can use an INSERT statement on a view that does not contain all the columns of the stored table on which it is based. For example, consider the EMPLOYEE table with none of the columns defined as NOT NULL. You could insert rows into the view PHONEBOOK even though it does not contain the MIDINIT, EDLEVEL or any other columns of the underlying table EMPLOYEE.

You can insert or update rows of a view in such a way that they do not satisfy the definition of the view. For example, the view PHONEBOOK is defined by the condition JOB <> 'PRES'. It would be possible to insert rows into PHONEBOOK having a value equal to 'PRES' in the JOB column. This insertion takes effect on the underlying table, EMPLOYEE, but the resulting rows are not visible in the view PHONEBOOK, because they do not satisfy the definition of PHONEBOOK. In fact, an update to PHONEBOOK that sets JOB='PRES' causes a row to “vanish” from PHONEBOOK (a cursor positioned on the row retains its position, but later scans through PHONEBOOK do not see this row). If you want to ensure that all rows inserted or updated are subsequently visible in the view, then define your view with 'WITH CHECK OPTION'.

However, the EMPLOYEE table does have columns defined as NOT NULL, and two of them (MIDINIT and EDLEVEL) are not available through the PHONE view. If you try to insert a row through the view, the system attempts to insert NULL values into all the EMPLOYEE columns that are “invisible” through the view. Because the MIDINIT and the EDLEVEL columns are not included in the view, and do not permit null values, the system does not permit the insertion through the view.

Be extremely careful when updating tables through views that may contain duplicate rows. For example, suppose a view JOBS is defined on the EMPLOYEE table containing only the columns WORKDEPT and JOB. Because EMPNO is not included in the view, and many employees may have the same job description, a user of the view cannot tell which EMPNO corresponds to a given row of the view. If the user positions a cursor on a row where JOB = 'CLERK', and then updates the current row of this cursor, a row of the stored EMPLOYEE table is updated. However, because there may be many clerks in the EMPLOYEE table, and the unique qualifier EMPNO is not part of the view, the user cannot control which employee is updated.

Dropping a View

Format

```
▶▶ DROP VIEW—view_name◀◀
```

The DROP VIEW statement drops the definition of the indicated view from the database. When you drop a view, the system also:

- Drops all other views defined in terms of the indicated view. (The underlying tables on which the views are defined are not affected.)
- Deletes all privileges on the dropped views from the authorization catalog tables.
- Marks invalid all packages that refer to the dropped views.

The invalid packages remain in the database until they are explicitly dropped by a DROP PACKAGE statement. When an invalid package is next invoked, the system attempts to regenerate it and restore its validity. However, if the program contains any SQL statement that refers to a dbspace, table, or view that has been dropped, that SQL statement returns an error code at run time.

If a DROP VIEW statement attempts to drop a view that is currently in use by another running logical unit of work, the statement is queued until that LUW ends.

Joining Tables

With joins, you can write a query against the combined data of two or more tables. (You can also join views.)

To join tables, follow these steps:

1. In the FROM clause, list all the tables you want to join.
2. In the WHERE clause, specify a *join condition* to express a relationship between the tables to be joined.

Note: The data types of the columns involved in the join condition do not have to be identical; however, they must be compatible. The join condition is evaluated the same way as any other search condition, and the same rules for comparisons apply. (These rules are discussed under “Using Expressions” on page 51.)

Joining Tables Using the Database Manager

The system forms all combinations of rows from the indicated tables. For each combination, it tests the join condition. If you do not specify a join condition, all combinations of rows from tables listed in the FROM clause are returned, even though the rows may be completely unrelated.

Performing a Simple Join Query

The join query in Figure 28 on page 67 finds the project number and the last name of the employees in department D11:

```

DECLARE C1 CURSOR FOR
SELECT PROJNO, LASTNAME
FROM EMPLOYEE, EMP_ACT
WHERE EMPLOYEE.EMPNO = EMP_ACT.EMPNO ← Join
AND WORKDEPT = 'D11'                                     Condition

ORDER BY PROJNO, LASTNAME

OPEN C1
FETCH C1 INTO :X, :Y
CLOSE C1

```

Figure 28. A Simple Join

The WHERE clause above expresses a join condition. If a row from one of the participating tables does not satisfy the join condition, that row does not appear in the result of the join. So, if a EMPNO in the EMPLOYEE table has no matching EMPNO in the EMP_ACT table (or if EMPNO in the EMP_ACT table has no matching EMPNO in the EMPLOYEE table), that row does not appear in your result.

Note: More than one table in a join may have a common column name. To identify exactly which column you are referring to, you must use the table name as a prefix, as in the example above. Unique column names do not require a table name prefix.

Here is the query result (based on the example tables):

PROJNO	LASTNAME
MA2111	BROWN
MA2111	BROWN
MA2111	LUTZ
MA2112	ADAMSON
MA2112	ADAMSON
MA2112	WALKER
MA2112	WALKER
MA2112	YOSHIMURA
MA2112	YOSHIMURA
MA2113	JONES
MA2113	JONES
MA2113	PIANKA
MA2113	SCOUTTEN
MA2113	YOSHIMURA

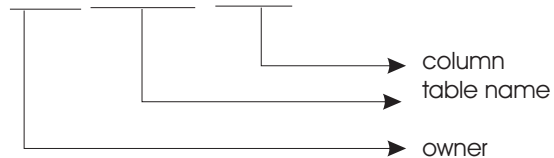
Joining Another User's Tables

If you are referring to another user's table, you must prefix the table name with the owner-name. If, for example, the tables in the query above belonged to JONES, you would write:

```

DECLARE C1 CURSOR FOR
SELECT PROJNO, LASTNAME
FROM JONES.EMPLOYEE, JONES.EMPACT
WHERE JONES.EMPLOYEE.EMPNO = JONES . EMP_ ACT . EMPNO
AND WORKDEPT = 'D11'
ORDER BY PROJNO, LASTNAME

```



```

OPEN C1
FETCH C1 INTO :X, :Y
CLOSE C1

```

Analyzing How a Join Works

When writing a join query, it is often helpful to mentally go through the query to see how SQL develops a JOIN.

For example, look at the previous select-statement. It refers to the EMPLOYEE and EMP_ACT tables. Joining the two tables will produce one table that contains all the columns in both tables.

Each EMPNO in the EMPLOYEE table is compared to every EMPNO in the EMP_ACT table. When the EMPNO column of both tables matches, a row is formed that contains the combined columns of the “matching” rows. Notice that the only column name that is common to both tables is EMPNO. If the name of this EMPNO column were different in each table, the EMPNO column of the result could have been called either name. This is because of the equality expressed in the join condition. In fact, the *select-list* could have specified EMPLOYEE.EMPNO instead of EMP_ACT.EMPNO, and identical results would have been produced.

Now consider what happens when the second part of the WHERE clause (AND WORKDEPT='D11') is applied.

The result is further reduced so that only the rows with a department name of D11 remain. The entire search condition is now satisfied. The system strips off the columns not specified in the *select-list*. This produces the query result previously shown.

Using VARCHAR and VARGRAPHIC within Join Conditions

If you are joining VARCHAR or VARGRAPHIC columns, trailing blanks are not used. For example, "JONES" and "JONES " match. If they were from two different EMPLOYEE tables joined on the LASTNAME column, they would form one row.

Using Nulls within Join Conditions

Like other predicates, a join condition is never satisfied by a null value. For example, if a row in the EMPLOYEE table and a row in the EMP_ACT table both have a null EMPNO, neither row will appear in the result of the join.

Joining a Table to Itself Using a Correlation Name

You can write a query in which you join a table to itself, by repeating the table name two or more times in the FROM clause. This tells the system that the join consists of combinations of rows from the same table. When you repeat the table name in the FROM clause, it is no longer unique. You must give one or both table names in the FROM clause a unique *correlation_name* to correctly designate the tables.

You use the correlation names to resolve column name ambiguities in the *select-list* and the WHERE clause. Rules for table designation are given at the end of this section.

For example, the following query finds the total of the values from the ACSTAFF column (PROJ_ACT table) for activities 60 and 70 for any project that contains both these activities:

```
DECLARE C1 CURSOR FOR
SELECT PA1.PROJNO, PA1.ACSTAFF + PA2.ACSTAFF
FROM   PROJ_ACT PA1, PROJ_ACT PA2
WHERE  PA1.PROJNO = PA2.PROJNO AND
       PA1.ACTNO = 60 AND PA2.ACTNO = 70
ORDER BY 1

OPEN C1
FETCH C1 INTO
       :PRONUM, :TOTAL
CLOSE C1
```

This type of join query can also be easily visualized. Each PROJNO in the PROJ_ACT table is compared to every other PROJNO in the PROJ_ACT table. When two rows with the same PROJNO are found, a row is formed. The new row contains the combined columns of the “matching” rows.

Now consider what happens when the second part of the WHERE clause (PA1.ACTNO = 60 AND PA2.ACTNO = 70) is applied.

The result is further reduced to only the rows with an ACTNO of 60 in the first ACTNO column and with an ACTNO of 70 in the second ACTNO column.

Finally, the system sorts the query by PROJNO and strips off the columns not specified in the *select-list*. This produces:

PROJNO	EXPRESSION 1	PROJNO	EXPRESSION 1
-----	-----	-----	-----
AD3111	2.30	AD3113	2.00
AD3111	1.30	AD3113	1.25
AD3111	2.00	AD3113	1.75
AD3111	1.00	AD3113	1.50
AD3112	1.50	AD3113	1.75
AD3112	1.25	AD3113	2.25
AD3112	1.75	AD3113	1.50
AD3112	1.00	AD3113	2.00
AD3112	1.25	AD3113	1.75
AD3112	1.00	AD3113	2.00
AD3112	1.50	AD3113	1.50
AD3112	0.75	AD3113	0.75
AD3112	1.50	AD3113	1.25
AD3112	1.25	AD3113	1.00
AD3112	1.75	AD3113	1.25
AD3112	1.00	MA2112	3.00
AD3112	1.75	MA2112	3.50
AD3112	1.50	MA2112	3.00
AD3112	2.00	MA2113	3.00
AD3112	1.25	MA2113	3.00

If the table is owned by another user, the table name must be qualified in the usual fashion. For example, here is how to write the above query if the owner of the PROJ_ACT table is SCOTT:

```

DECLARE C1 CURSOR FOR
SELECT PA1.PROJNO, PA1.ACSTAFF + PA2.ACSTAFF
FROM SCOTT.PROJ_ACT PA1, SCOTT.PROJ_ACT PA2
WHERE PA1.PROJNO = PA2.PROJNO AND
      PA1.ACTNO = 60 AND PA2.ACTNO = 70
ORDER BY 1
OPEN C1
FETCH C1 INTO
      :PRONUM, :TOTAL
CLOSE C1

```

Rules for Table Designation

1. Only exposed table names and correlation names in the FROM clause can be referenced in other clauses.

An exposed table name is one that is not followed by a *correlation_name* (for example, PROJECT). A nonexposed table name is a table name which is followed by a *correlation_name* (for example, PROJECT P). In the latter example, PROJECT has no scope in the query and cannot be referenced; the table designator in this case is P.

2. Exposed table names in the FROM clause must be different from each other.
3. Correlation names in the FROM clause must be different from each other and different from any exposed table names.

These rules are illustrated here:

```
SELECT EMPLOYEE.LASTNAME FROM EMPLOYEE E
```

← Incorrect

The above query is not allowed. EMPLOYEE is a nonexposed table name and cannot be used to qualify column LASTNAME.

```
SELECT EMPLOYEE.LASTNAME FROM EMPLOYEE E, EMPLOYEE
```

← Correct

The above query is allowed. The second table in the FROM clause can be designated by the exposed table name EMPLOYEE. There is no ambiguity or conflict with the table name EMPLOYEE in the first table of the FROM clause, because that is a nonexposed table name.

Imposing Limits on Join Queries

The example of a simple join query in Figure 28 on page 67 had only one join condition relating the values of EMPNO in two tables.

Note: A query has the following limitations:

- You can join up to 15 tables
- A query can have up to 40 join conditions (predicates)
- The WHERE clause of a query may have a total of up to 200 conditions (predicates)

For more information on these limits, see the section on 'SQL Limits' in the *DB2 Server for VSE & VM SQL Reference* manual.

Using SELECT * In a Join

The notation SELECT * in a join query means “select all the columns of the first table, followed by all the columns of the second table, and so on.” You can also use the notation SELECT T1.* to select all the columns of the table T1. However, it is not recommended that you use either SELECT * or SELECT T1.* for join queries written in programs because if someone adds a new column to the first table in the join (by an ALTER TABLE statement), the columns of the second table are no longer delivered into the correct host variables. To avoid this problem, use a *select-list* in which all the columns are specifically listed.

Grouping the Rows of a Table

The *DB2 Server for VSE & VM SQL Reference* manual shows how to apply the column functions (SUM, AVG, MIN, MAX, and COUNT) to a table. However, you can apply these functions only to particular columns in rows that satisfy a search condition. For example, the following statement finds the average number of employees for all occurrences of project number AD3111 in the PROJ_ACT table:

```

SELECT AVG(ACSTAFF)
FROM PROJ_ACT
WHERE PROJNO = 'AD3111'

```

In contrast, the *grouping* feature of the database manager permits you to conceptually divide a table into groups of rows with matching values in one or more columns. You can then apply a function to each group. For example, to find the average number of employees for each project in the PROJ_ACT table:

```

SELECT PROJNO,AVG(ACSTAFF)
FROM PROJ_ACT
GROUP BY PROJNO
ORDER BY PROJNO

```

The query yields this result based on the sample table PROJ_ACT:

PROJNO	AVG(ACSTAFF)
-----	-----
AD3100	0.50000000000000000000000000000000
AD3110	1.00000000000000000000000000000000
AD3111	0.93571428571428571428571428571428
AD3112	0.62272727272727272727272727272727
AD3113	0.84615384615384615384615384615384
IF1000	0.60000000000000000000000000000000
IF2000	0.55000000000000000000000000000000
MA2100	0.75000000000000000000000000000000
MA2110	1.00000000000000000000000000000000
MA2111	1.00000000000000000000000000000000
MA2112	1.21428571428571428571428571428571
MA2113	1.07142857142857142857142857142857
OP1000	0.25000000000000000000000000000000
OP1010	2.50000000000000000000000000000000
OP2000	0.75000000000000000000000000000000
OP2010	1.00000000000000000000000000000000
OP2011	0.50000000000000000000000000000000
OP2012	0.50000000000000000000000000000000
OP2013	0.50000000000000000000000000000000
PL2100	1.00000000000000000000000000000000

One or more column functions can be applied to the groups. The following query finds the maximum, minimum, and average salary for each department, along with the count of the number of rows in each group (the column function COUNT(*) evaluates to the number of rows in the group):

```

SELECT WORKDEPT, MAX(SALARY), MIN(SALARY), AVG(SALARY), COUNT(*)
FROM EMPLOYEE
GROUP BY WORKDEPT

```

Using VARCHAR and VARGRAPHIC within Groups

If you are grouping a VARCHAR or VARGRAPHIC column, trailing blanks are ignored. For example, if a select-statement was grouped by DESCRIPTION, "BOLT" and "BOLT " would match. They would be placed in the same group.

Using Nulls within Groups

If you are grouping columns that return null values, the null values are grouped in those columns. The null values may be returned because of undefined column values or arithmetic exception errors.

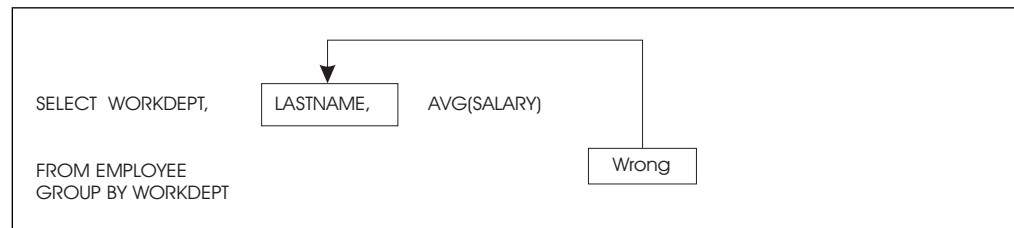
If you have defined a VIEW that contains a GROUP BY clause, the view columns named in the GROUP BY have the same nullability as the corresponding base table columns.

Using Select-Lists in Grouped Queries

When you use the GROUP BY clause in a query, the database manager returns only one result row for each group. The *select-list* of such a query can contain only:

- GROUP BY columns
- Column functions.

For example, this statement is incorrect:



You cannot include LASTNAME in the *select-list* because LASTNAME does not occur in the GROUP BY clause, and is not the operand of a column function. Aside from breaking language rules, the above statement is incorrect because a department may have many employees. It is as though you were asking the system to return multiple values to the same variable at the same time.

Using a WHERE Clause with a GROUP BY Clause

A grouping query can have a standard WHERE clause that eliminates non-qualifying rows before the groups are formed and the column functions are computed. Write the WHERE clause *before* the GROUP BY clause. For example:

```
SELECT WORKDEPT, AVG(SALARY)
FROM EMPLOYEE
WHERE HIREDATE > '1970-01-01'
GROUP BY WORKDEPT
```

Using the HAVING Clause

You can apply a qualifying condition to groups so that the system returns a result only for the groups that satisfy the condition, by including a HAVING clause *after* the GROUP BY clause. A HAVING clause can contain one or more group-qualifying predicates connected by ANDs and ORs. Each group-qualifying predicate compares a property of the group such as AVG(ACSTAFF) with one of the following:

1. Another property of the group (for example, HAVING AVG(ACSTAFF) > 2 * MIN(ACSTAFF))
2. A constant (for example, HAVING AVG(ACSTAFF) > 1.00)

3. A host variable (for example, `HAVING AVG(ACSTAFF) > :LIMIT`).

For example, the following query finds the average mean number of employees for projects having more than three activities:

```
SELECT PROJNO,AVG(ACSTAFF)
   FROM PROJ_ACT
  GROUP BY PROJNO
  HAVING COUNT(*) > 3
  ORDER BY PROJNO
```

You can specify `DISTINCT` as part of the argument of a column function in the `HAVING` clause, because `DISTINCT` eliminates duplicate values before a function is applied. Thus, `COUNT(DISTINCT PROJNO)` computes the number of different project numbers. You cannot use `DISTINCT` in both the *select-list* and `HAVING` clause; you can use it only once in a query.

It is possible (though unusual) for a query to have a `HAVING` clause but no `GROUP BY` clause. In this case, the system treats the entire table as one group. Because the table is treated as a single group, you can have at most one result row. If the `HAVING` condition is true for the table as a whole, the selected result (which must consist entirely of column functions) is returned; otherwise the “not found” code (`SQLCODE = 100` and `SQLSTATE='02000'`) is returned.

Combining Joins

This section discusses the `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses of the `select-statement`.

You can use the various query techniques together in any combination. A query can join two or more tables and can also have a `WHERE` clause, a `GROUP BY` clause, a `HAVING` clause, and, if defined in a cursor, an `ORDER BY` clause. The sequence of application for these clauses is listed below:

1. Conceptually, all possible combinations of rows from the listed tables are formed.
2. The `WHERE` clause, which may contain join conditions, is applied to filter the rows of the conceptual table.
3. The `GROUP BY` clause is applied to form groups from the surviving rows.
4. The `HAVING` clause is applied to filter the groups. Only the surviving groups will return a result.
5. The *select-list* expressions are evaluated.
6. The `ORDER BY` clause determines the order in which the query result is returned.

Illustrating Grouping with an Exercise

By now you may be wondering when you need to use which feature. Consider this problem:

Write a query that returns:

- *The department number*
- *The manager's employee number*
- *The total number of activities for all the projects in the department*

- *The sum of the estimated mean number of employees needed to staff the activities for all the projects in the department.*

Consider only projects that are estimated to end after January, 1 1994, and only include departments with more than two activities. Finally, order the result by department name.

The first thing that you must do is to find in the example tables the names of the columns that contain the requested information, so that you can create a *select-list*:

- “department number” is the DEPTNO column of the DEPARTMENT table.
- “manager's employee number” is the MGRNO column of the DEPARTMENT table.
- “activities” is the ACTNO column of the PROJ_ACT table, but the problem requests the *total number of activities for all the projects in a department*, so you must include the column function COUNT(*) in the *select-list*.

Note: You need the total number of activities for a particular department; this means that the query will have to group by department.

- “estimated mean number of employees needed to staff the activities” implies the ACSTAFF column of the PROJ_ACT table. However, the problem requests **The sum of the estimated mean number of employees needed to staff the activities for all the projects in the department.** So you must include the column function SUM in the *select-list*; this means that the query will have to group by department.

Note: The columns DEPTNO and MGRNO (from the DEPARTMENT table) and ACSTAFF (from the PROJ_ACT table) come from different tables so you will need a join. However, the DEPARTMENT, and PROJ_ACT tables do not have a common column. To join them, you will have to use the PROJECT table in a three-table join. PROJECT contains both the DEPTNO column of the DEPARTMENT table and the PROJNO column of the PROJ_ACT table.

First, define the cursor(s) to be used in your program:

```
DECLARE C1 CURSOR FOR
```

Now write a SELECT clause:

```
SELECT DEPARTMENT.DEPTNO, MGRNO, SUM(ACSTAFF), COUNT(*)
```

Note: Since a DEPTNO column appears in both the DEPARTMENT and the PROJECT tables, you must qualify which table it is from.

Write a FROM clause that lists the three tables used in the join:

```
FROM DEPARTMENT, PROJECT, PROJ_ACT
```

You must include a WHERE clause because of the join condition; one line to join the DEPARTMENT table to the PROJECT table, and one to join the PROJECT table to the PROJ_ACT table:

```
WHERE DEPARTMENT.DEPTNO = PROJECT.DEPTNO
AND PROJECT.PROJNO = PROJ_ACT.PROJNO
```

However, the problem states that only projects that are estimated to end on or after January 1, 1994 should be considered. This condition needs to be added to the WHERE clause:

```
AND PRENDATE >= '1994-01-01'
```

Note that PRENDATE is a column in the PROJ_ACT table and is unique among all the column names of the joined tables, so it does not have to be qualified. So far, the SQL statement is:

```
DECLARE C1 CURSOR FOR
SELECT DEPARTMENT.DEPTNO, MGRNO, SUM(ACSTAFF), COUNT(*)
FROM DEPARTMENT, PROJECT, PROJ_ACT
WHERE DEPARTMENT.DEPTNO = PROJECT.DEPTNO
AND PROJECT.PROJNO = PROJ_ACT.PROJNO
AND PRENDATE >= '1994-01-01'
```

It is now necessary to group by DEPTNO to find the sum for each part, but MGRNO is also in the *select-list*, so it must be listed in the GROUP BY clause (recall the rules for grouping). Including MGRNO in the GROUP BY clause does not affect the formation of the groups, however, because MGRNO is a property of a given DEPTNO. The GROUP BY clause is:

```
GROUP BY DEPARTMENT.DEPTNO, MGRNO
```

Note: You can group by PROJECT.DEPTNO if you choose, because of the equality expressed between DEPARTMENT.DEPTNO and PROJECT.DEPTNO in the join condition. If you use PROJECT.DEPTNO in the GROUP BY clause, however, you must also use it in the *select-list*.

If the table name is fully qualified in the FROM clause, it is good practice to fully qualify it in the whole statement.

The problem requires that the departments included in the query have at least two activities for all the projects in the department; a HAVING clause is needed to filter out the unwanted groups:

```
HAVING COUNT(*) > 2
```

To have the system return the results in DEPTNO order, type:

```
DECLARE C1 CURSOR FOR
SELECT DEPARTMENT.DEPTNO, MGRNO, SUM(ACSTAFF), COUNT(*)
FROM DEPARTMENT, PROJECT, PROJ_ACT
WHERE DEPARTMENT.DEPTNO = PROJECT.DEPTNO
AND PROJECT.PROJNO = PROJ_ACT.PROJNO
AND PRENDATE >= '1994-01-01'
GROUP BY DEPARTMENT.DEPTNO, MGRNO
HAVING COUNT(*) > 2
ORDER BY 1
```

Now you must position the cursor and identify the corresponding host variables used in your program:

```

OPEN C1
FETCH C1 INTO :DEPT, :MGRN, :TOTSTAFF, :NUMACT
CLOSE C1

```

By incorporating the FETCH statement in a suitable host program loop along with an appropriate output command, this query produces the following result:

DEPTNO	MGRNO	SUM(ACSTAFF)	COUNT(EXPRESSION 1)
C01	000030	5.75	10
D01	?	2.00	3
D21	000070	25.40	32
E21	000100	4.00	7

Nesting Queries

In all previous queries, the WHERE clause contained search conditions that the database manager used to choose rows for computing expressions in the select-list. A query can refer to a value or set of values computed by another query (called a *subquery*).

Consider this query which finds all the activities for project IF1000:

```

SELECT ACTNO, ACSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'

```

Suppose that you want to modify the query so it finds the activities for project IF1000 whose estimated mean number of employees is greater than the minimum estimated mean for that project.

The problem involves two queries:

1. Find the minimum estimated mean number of employees for project IF1000

```
SELECT
    MIN (ACSTAFF)
INTO :MINSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
```

2. Find quotations for project number IF1000 find the estimated mean number of employees needed to staff the activity.

```
DECLARE C1 CURSOR FOR
SELECT ACTNO, ACSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
```

```
AND ACSTAFF >
```

?

```
OPEN C1
FETCH C1 INTO :AN, :AS
CLOSE C1
```

A pseudocode solution for the problem is as follows:

```
EXEC SQL SELECT MIN (ACSTAFF)
INTO :MINSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
```

```
EXEC SQL DECLARE C1 CURSOR FOR
SELECT ACTNO, ACSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
AND ACSTAFF > :MINSTAFF
```

```
EXEC SQL OPEN C1
EXEC SQL FETCH C1 INTO :AN, :AS
DO WHILE (SQLCODE=0)
    DISPLAY (AN, AS)
EXEC SQL FETCH C1 INTO :AN, :AS
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1
```

Initialize ACSTAFF

Declare cursor using a subquery that retrieves quotations

Retrieve quotations

You can arrive at the same result by using a single query with a subquery. Subqueries must be enclosed in parentheses, and may appear in a WHERE clause or a HAVING clause. The result of the subquery is substituted directly into the

outer-level predicate in which the subquery appears; thus, there must not be an INTO clause in a subquery. For example, this query solves the above problem:

```

DECLARE C1 CURSOR FOR
SELECT ACTNO, ACSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
AND ACSTAFF >
    (SELECT MIN(ACSTAFF)
     FROM PROJ_ACT
     WHERE PROJNO = 'IF1000')

OPEN C1
FETCH C1 INTO :AN, :AS
CLOSE C1

```

The example subquery above is indented for ease of reading. Remember, however, that the syntax of SQL is fully linear and no syntactic meaning is carried by indentation or by breaking a query into several lines.

By using a subquery, the pseudocode is simplified:

```

EXEC SQL DECLARE C1 CURSOR FOR
SELECT ACTNO, ACSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
AND ACSTAFF >
    (SELECT MIN(ACSTAFF)
     FROM PROJ_ACT
     WHERE PROJNO = 'IF1000')

EXEC SQL OPEN C1
EXEC SQL FETCH C1 INTO :AN, :AS
DO WHILE (SQLCODE=0)
    DISPLAY (AN, AS)
    EXEC SQL FETCH C1 INTO :AN, :AS
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1

```

The subquery above returns a single value MIN(ACSTAFF) to the outer-level query. Subqueries can return either a single value, no value, or a set of values; each variation has different considerations. In any case, a subquery must have only a single column or expression in its *select-list*, and must not have an ORDER BY clause.

Returning a Single Value: If a subquery returns a single value, as the one subquery above did, you can use it on the right side of any predicate in the WHERE clause or HAVING clause.

Returning No Value: If a subquery returns no value (an empty set), the outer-level predicate containing the subquery evaluates to the unknown truth-value.

Returning Many Values: If a subquery returns more than one value, you must modify the comparison operators in your predicate by attaching the suffix ALL, ANY, or SOME. These suffixes determine how the set of values returned is to be treated in the outer-level predicate. The > comparison operator is used as an example (the remarks below apply to the other operators as well):

expression > (subquery)

denotes that the subquery must return one value at most (otherwise an error condition results). The predicate is true if the given column is greater than the value returned by the subquery.

expression >ALL (subquery)

denotes that the subquery may return a set of zero, one, or more values. The predicate is true if the given column is greater than each individual value in the returned set. If the subquery returns no values, the predicate is true.

expression >ANY (subquery)

denotes that the subquery may return a set of zero, one, or more values. The predicate is true if the given column is greater than at least one of the values in the set. If the subquery returns no values, the predicate is false.

expression >SOME (subquery)

SOME and ANY are synonymous.

The following example uses a > ALL comparison to find those projects with activities whose estimated mean number of employees is greater than all of the corresponding numbers for project AD3111:

```
DECLARE C1 CURSOR FOR
SELECT PROJNO, ACTNO
FROM PROJ_ACT
WHERE ACSTAFF > ALL
      (SELECT ACSTAFF
       FROM PROJ_ACT
       WHERE PROJNO = 'AD3111')

OPEN C1
FETCH C1 INTO :PN, :AN
CLOSE C1
```

Using the IN Predicate with a Subquery

Your query can also use the operators IN and NOT IN when a subquery returns a set of values. For example, the following query lists the surnames of employees responsible for projects MA2100 and OP2012:

```

DECLARE C1 CURSOR FOR
SELECT LASTNAME
FROM EMPLOYEE
WHERE EMPNO IN
      (SELECT RESPEMP
       FROM PROJECT
       WHERE PROJNO = 'MA2100'
       OR PROJNO = 'OP2012')

OPEN C1
FETCH C1 INTO :LNAME
CLOSE C1

```

The subquery is evaluated once, and the resulting list is substituted directly into the outer-level query. For example, if the subquery above selects employee numbers 60 and 330, the outer-level query is evaluated as if its WHERE clause were:

```
WHERE EMPNO IN (60, 330)
```

The list of values returned by the subquery can contain zero, one, or more values. The operator IN is equivalent to =ANY, and NOT IN is equivalent to <>ALL.

Considering Other Subquery Issues

A subquery can contain GROUP BY or HAVING clauses. If it is linked by an unmodified comparison operator such as = or >, the subquery may return one group. If it is linked by a modified comparison operator ALL, ANY, or SOME, [NOT] IN, or [NOT] EXISTS, it may return more than one group.

A subquery may include a join, a grouping, or one or more inner-level subqueries. You may include many subqueries in the same outer-level query, each in its own predicate and enclosed in parentheses.

Executing Subqueries Repeatedly: Correlation

In all the examples of subqueries above, the subquery is evaluated only once and the resulting value or set of values is substituted into the outer-level predicate. For example, recall this query from the previous section:

```

DECLARE C1 CURSOR FOR
SELECT ACTNO, ACSTAFF
FROM PROJ_ACT
WHERE PROJNO = 'IF1000'
AND ACSTAFF >
      (SELECT MIN(ACSTAFF)
       FROM PROJ_ACT
       WHERE PROJNO = 'IF1000')

```

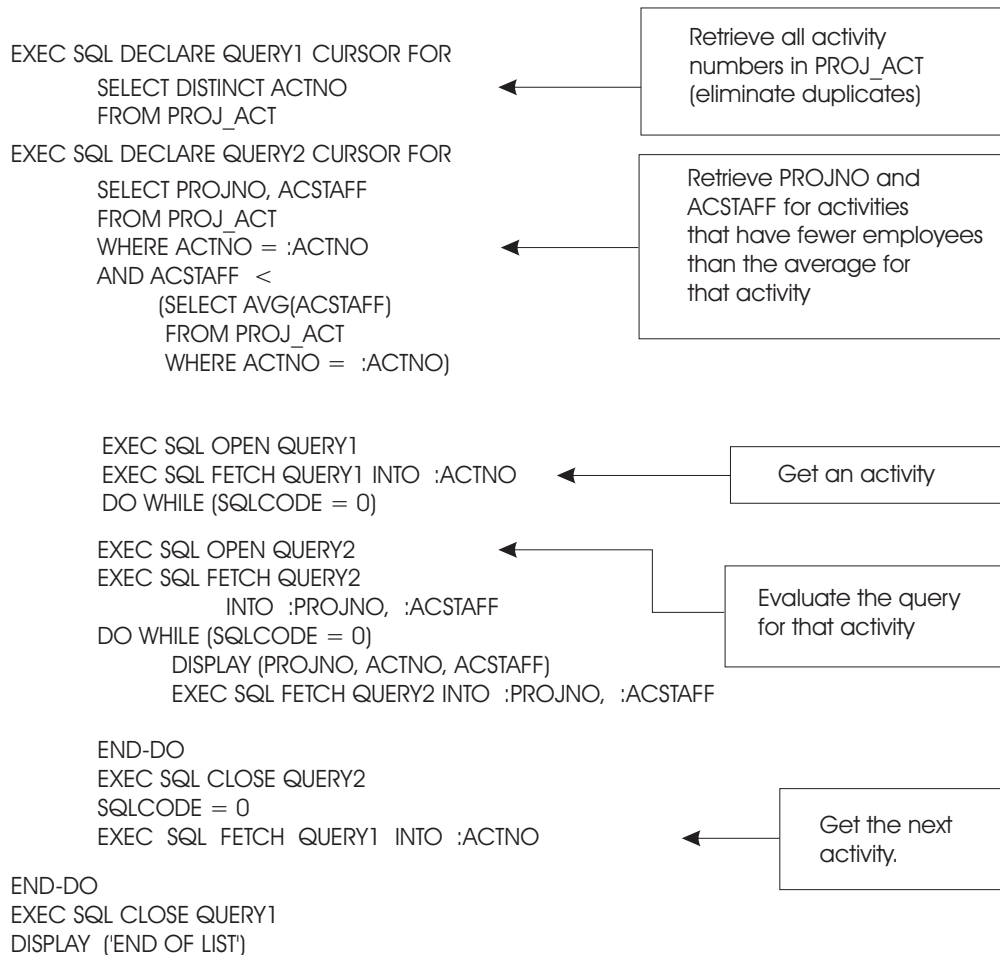
This query finds the activities for project IF1000 whose estimated mean number of employees is greater than the minimum estimated mean for that project. Now consider the following problem:

Find the project and activity numbers for activities that have an estimated mean number of employees that is less than the average estimated mean for that activity as calculated across all projects.

The subquery needs to be evaluated once for every activity number. You can do this by using the *correlation* capability of SQL, which permits you to write a subquery that is executed *repeatedly*, once for each row of the table identified in the outer-level query. This type of “correlated subquery” computes some property of each row of the outer-level table that is needed to evaluate a predicate in the subquery.

In the first query, the subquery was evaluated *once* for a particular project; in the new problem, it must be evaluated once for *every* activity. One way to solve the problem is to place the query in a cursor definition and open the cursor once for each different activity. The activities are determined by using a separate cursor.

Here is a pseudocode solution:



By using a correlated subquery, you can let the system do the work for you and reduce the amount of code you need to write.

Writing a Correlated Subquery

To write a query with a correlated subquery, you use the same basic format as an ordinary outer query with a subquery. However, in the FROM clause of the outer query, just after the table name, you place a *correlation_name*. (See “Joining a Table to Itself Using a Correlation Name” on page 69 for more information on correlation names.) The subquery may then contain column references qualified by the *correlation_name*. For example, if X is a *correlation_name*, then “X.ACTNO” means “the ACTNO value of the current row of the table in the outer query.” The subquery is (conceptually) reevaluated for each row of the table in the outer query.

The following query solves the problem presented earlier. That is, it finds the project and activity numbers for activities that have an estimated mean number of employees that is less than the average estimated mean for that activity, as calculated across all projects.

```
SELECT PROJNO,ACTNO,ACSTAFF
FROM PROJ_ACT X
WHERE ACSTAFF < (SELECT AVG(ACSTAFF)
FROM PROJ_ACT
WHERE ACTNO = X.ACTNO)
```

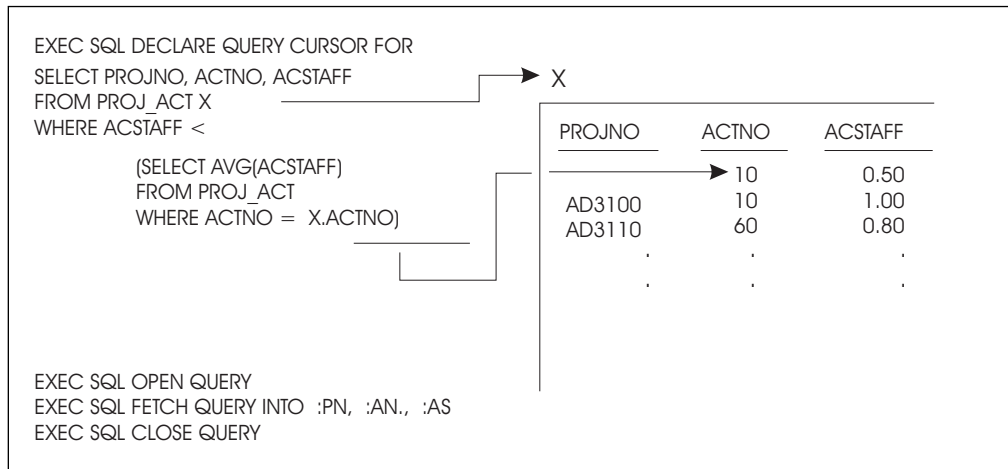
The pseudocode for the correlated subquery solution is:

```
EXEC SQL DECLARE QUERY CURSOR FOR
SELECT PROJNO,ACTNO,ACSTAFF
FROM PROJ_ACT X
WHERE ACSTAFF < (SELECT AVG(ACSTAFF)
FROM PROJ_ACT
WHERE ACTNO = X.ACTNO)
EXEC SQL OPEN QUERY
EXEC SQL FETCH QUERY INTO :PROJNO, :ACTNO, :ACSTAFF
DO WHILE (SQLCODE=0)
DISPLAY (PROJNO, ACTNO, ACSTAFF)
EXEC SQL FETCH QUERY INTO :PROJNO, :ACTNO, :ACSTAFF
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE QUERY
```

How the Database Manager Does Correlation

Conceptually, the query is evaluated as follows:

1. PROJ_ACT, the table identified with the *correlation_name* X, is placed to the side for reference. Let this table be called X, because it is the *correlation table*.
2. The system identifies X.ACTNO with the X table, and uses the values in that column to evaluate the query. (The entire query is evaluated once for every ACTNO in the X table.)

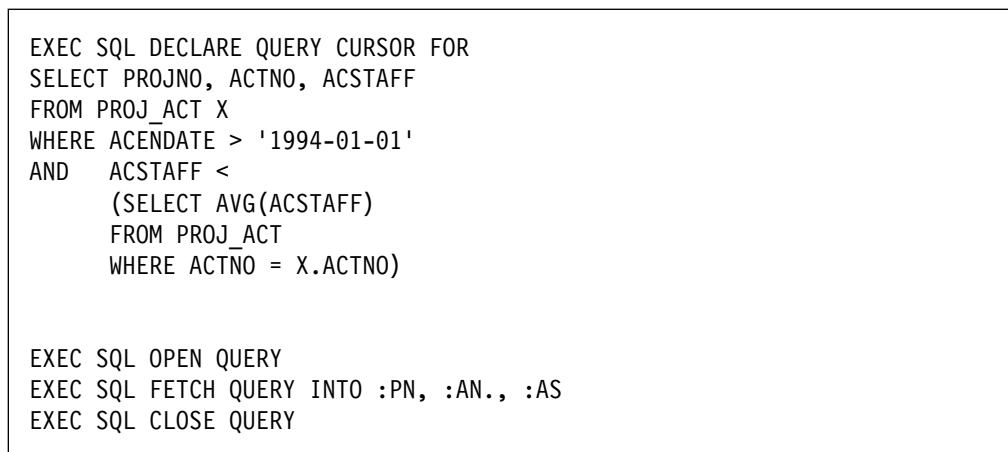


Note: ACTNO = X.ACTNO is not used in the WHERE clause of the outer-level query as it was in the uncorrelated subquery, because the system keeps track of the X.ACTNO for which it is evaluating the query.

Suppose another condition is added to the problem:

*Find the project and activity numbers **for activities that have an estimated end date after January 1, 1994 and** have an estimated mean number of employees that is less than the average estimated mean for that activity.*

The new query is:



The X table in this query is slightly different. Conceptually, whenever there are other conditions besides the one containing the subquery, they are applied to the *correlation table* first. The X table that is derived from the PROJ_ACT table is:

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3111	80	1.25	1991-04-15	1994-01-15
MA2112	70	1.50	1991-02-15	1994-02-01
MA2113	70	2.00	1991-04-01	1994-12-15
MA2113	80	1.50	1991-09-01	1994-02-01
OP1010	130	4.00	1991-01-01	1994-02-01

Only rows with an ACENDATE greater than '1994-01-01' are included in this "correlation table".

The values 70, 80, and 130 are used for X.ACTNO. Similarly, if you include a GROUP BY clause in the outer-level query, that grouping is applied to the conceptual correlation table first. Thus, if you use a correlated subquery in a HAVING clause, it is evaluated once per *group* of the conceptual table (as defined by the outer-level query's GROUP BY clause). When you use a correlated subquery in a HAVING clause, the correlated column-reference in the subquery must be a property of each group (that is, must be either the *group-identifying* column or another column used with a column function).

The use of a column function with a correlated reference in a subquery is called a *correlated function*. The argument of a correlated function must be exactly one correlated column (for example, X.ACSTAFF), not an expression. A correlated function may specify the DISTINCT option; for example: COUNT(DISTINCT X.ACTNO). If so, the DISTINCT counts as the single permitted DISTINCT specification for the outer-level query-block (remember that each query-block may use DISTINCT only once). For information on query-block, refer to the *DB2 Server for VSE Database Administration* manual.

Illustrating a Correlated Subquery

When would you want to use a correlated subquery? The use of a column function is sometimes a clue. Consider this problem:

List the employees whose level of education is higher than the average for their department.

First you must determine the *select-list* items. The problem says to "List the employees". This implies that the query should return something to identify the employees. LASTNAME from the EMPLOYEE table should be sufficient. The problem also discusses the level of education (EDLEVEL) and the employees' departments (WORKDEPT). While the problem does not explicitly ask for these columns, including them in the *select-list* will help illustrate the solution. A part of the query can now be constructed:

```
SELECT LASTNAME, WORKDEPT, EDLEVEL
FROM EMPLOYEE
```

Next, a search condition (WHERE clause) is needed. The problem statement says, "...whose level of education is higher than the average for that employee's department". This means that for every employee in the table, the average education level for that employee's department must be computed. This statement fits the description of a correlated subquery. Some property (average level of

education of the current employee's department) is being computed for each row. A *correlation_name* is needed on the EMPLOYEE table:

```
SELECT LASTNAME, WORKDEPT, EDLEVEL
FROM EMPLOYEE Y
```

The subquery needed is simple; it computes the average level of education for each department:

<pre>SELECT AVG(EDLEVEL) FROM EMPLOYEE WHERE WORKDEPT = Y.WORKDEPT</pre>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <p>This clause tells the database manager to compute the subquery once for each employee in the outer-level query table.</p> </div>
--	---

The complete SQL statement is:

<pre>SELECT LASTNAME, WORKDEPT, EDLEVEL FROM EMPLOYEE Y WHERE EDLEVEL > (SELECT AVG(EDLEVEL) FROM EMPLOYEE WHERE WORKDEPT = Y.WORKDEPT)</pre> <p>This will produce the following:</p> <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">LASTNAME</th> <th style="text-align: left;">WORKDEPT</th> <th style="text-align: left;">EDLEVEL</th> </tr> <tr> <th colspan="3" style="border-top: 1px dashed black; border-bottom: 1px dashed black;"></th> </tr> </thead> <tbody> <tr><td>HAAS</td><td>A00</td><td>18</td></tr> <tr><td>KWAN</td><td>C01</td><td>20</td></tr> <tr><td>PULASKI</td><td>D21</td><td>16</td></tr> <tr><td>HENDERSON</td><td>E11</td><td>16</td></tr> <tr><td>LUCCHESI</td><td>A00</td><td>19</td></tr> <tr><td>PIANKA</td><td>D11</td><td>17</td></tr> <tr><td>SCOUTTEN</td><td>D11</td><td>17</td></tr> <tr><td>JONES</td><td>D11</td><td>17</td></tr> <tr><td>LUTZ</td><td>D11</td><td>18</td></tr> <tr><td>MARINO</td><td>D21</td><td>17</td></tr> <tr><td>JOHNSON</td><td>D21</td><td>16</td></tr> <tr><td>SCHNEIDER</td><td>E11</td><td>17</td></tr> <tr><td>MEHTA</td><td>E21</td><td>16</td></tr> <tr><td>GOUNOT</td><td>E21</td><td>16</td></tr> </tbody> </table>	LASTNAME	WORKDEPT	EDLEVEL				HAAS	A00	18	KWAN	C01	20	PULASKI	D21	16	HENDERSON	E11	16	LUCCHESI	A00	19	PIANKA	D11	17	SCOUTTEN	D11	17	JONES	D11	17	LUTZ	D11	18	MARINO	D21	17	JOHNSON	D21	16	SCHNEIDER	E11	17	MEHTA	E21	16	GOUNOT	E21	16
LASTNAME	WORKDEPT	EDLEVEL																																														
HAAS	A00	18																																														
KWAN	C01	20																																														
PULASKI	D21	16																																														
HENDERSON	E11	16																																														
LUCCHESI	A00	19																																														
PIANKA	D11	17																																														
SCOUTTEN	D11	17																																														
JONES	D11	17																																														
LUTZ	D11	18																																														
MARINO	D21	17																																														
JOHNSON	D21	16																																														
SCHNEIDER	E11	17																																														
MEHTA	E21	16																																														
GOUNOT	E21	16																																														

Suppose that instead of listing the employee's department number, you list the department name. A glance at the sample tables will tell you that the information you need (DEPTNAME) is in a separate table (DEPARTMENT). The outer-level query that defines a correlation variable can also be a join query.

When you use joins in an outer-level query, list the tables to be joined in the FROM clause, and place the *correlation_name* next to one of these table names.

To modify the query to list the department's name instead of the number, replace WORKDEPT by DEPTNAME in the *select-list*. The FROM clause must now also

include the DEPARTMENT table, and the WHERE clause must express the appropriate join condition.

This is the modified query:

```
SELECT LASTNAME, DEPTNAME, EDLEVEL
       FROM EMPLOYEE Y, DEPARTMENT
       WHERE Y.WORKDEPT = DEPARTMENT.DEPTNO
       AND EDLEVEL >
          (SELECT AVG(EDLEVEL)
           FROM EMPLOYEE
           WHERE WORKDEPT = Y.WORKDEPT)
```

This will produce the following:

LASTNAME	DEPTNAME	EDLEVEL
HAAS	SPIFFY COMPUTER SERVICE DIV.	18
LUCCHESI	SPIFFY COMPUTER SERVICE DIV.	19
KWAN	INFORMATION CENTER	20
PIANKA	MANUFACTURING SYSTEMS	17
SCOUTTEN	MANUFACTURING SYSTEMS	17
JONES	MANUFACTURING SYSTEMS	17
LUTZ	MANUFACTURING SYSTEMS	18
PULASKI	ADMINISTRATION SYSTEMS	16
MARINO	ADMINISTRATION SYSTEMS	17
JOHNSON	ADMINISTRATION SYSTEMS	16
HENDERSON	OPERATIONS	16
SCHNEIDER	OPERATIONS	17
MEHTA	SOFTWARE SUPPORT	16
GOUNOT	SOFTWARE SUPPORT	16

The above examples show that the *correlation_name* used in a subquery must be defined in the FROM clause of some query that contains the correlated subquery. However, this containment may involve several levels of nesting. Suppose that some departments have only a few employees and therefore their average education level may be misleading. You might decide that in order for the average level of education to be a meaningful number to compare an employee against, there must be at least five employees in a department. The new statement of the problem is:

List the employees whose level of education is higher than the average for that employee's department. Only consider departments with at least five employees.

The problem implies another subquery because, for each employee in the outer-level query, the total number of employees in that persons department must be counted:

```
SELECT COUNT(*)
       FROM EMPLOYEE
       WHERE WORKDEPT = Y.WORKDEPT
```

Only if the count is greater than or equal to 5 is an average to be computed:

```
SELECT AVG(EDLEVEL)
FROM EMPLOYEE
WHERE WORKDEPT = Y.WORKDEPT
AND 5 <=
(SELECT COUNT(*)
FROM EMPLOYEE
WHERE WORKDEPT = Y.WORKDEPT)
```

Finally, only those employees whose level of education is greater than the average for that department are included:

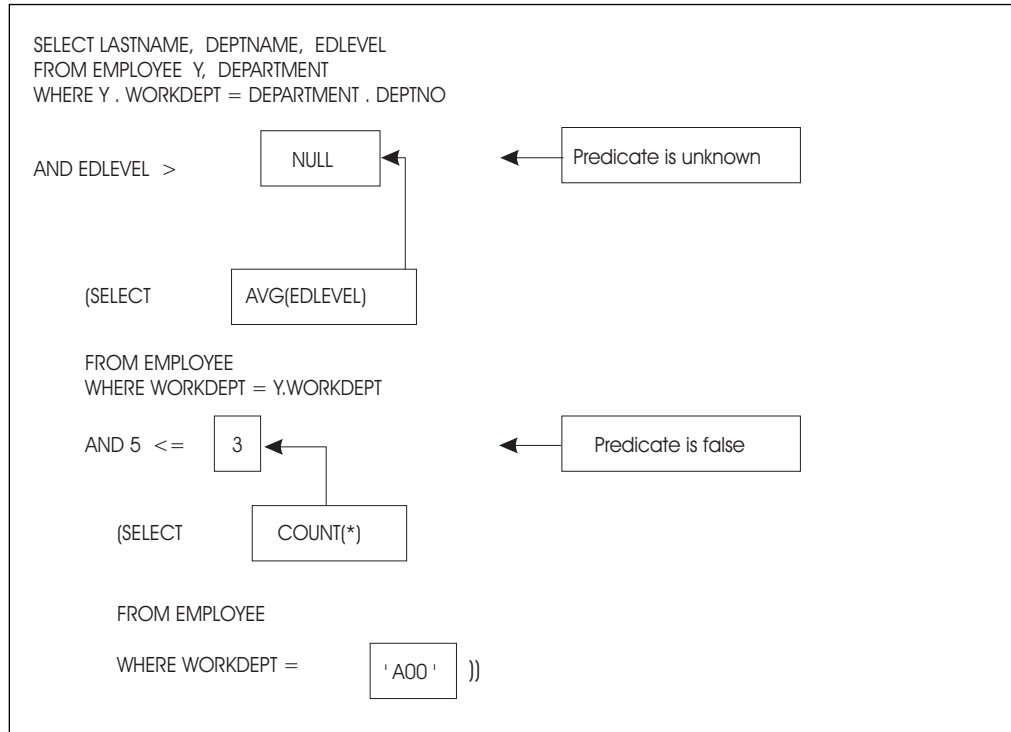
```
SELECT LASTNAME, DEPTNAME, EDLEVEL
FROM EMPLOYEE Y, DEPARTMENT
WHERE Y.WORKDEPT = DEPARTMENT.DEPTNO
AND EDLEVEL >
(SELECT AVG(EDLEVEL)
FROM EMPLOYEE
WHERE WORKDEPT = Y.WORKDEPT
AND 5 <=
(SELECT COUNT(*)
FROM EMPLOYEE
WHERE WORKDEPT = Y.WORKDEPT));
```

This will produce the following:

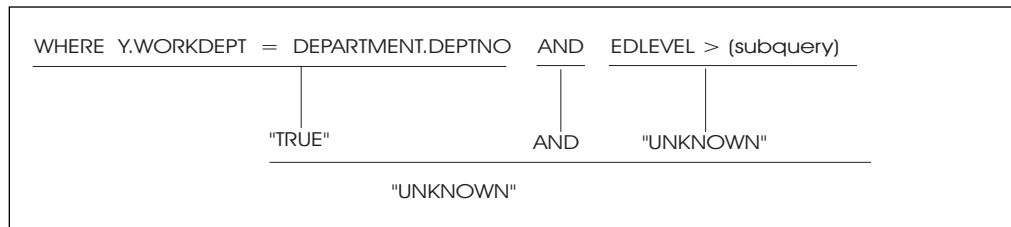
LASTNAME	DEPTNAME	EDLEVEL
PIANKA	MANUFACTURING SYSTEMS	17
SCOTTEN	MANUFACTURING SYSTEMS	17
JONES	MANUFACTURING SYSTEMS	17
LUTZ	MANUFACTURING SYSTEMS	18
PULASKI	ADMINISTRATION SYSTEMS	16
MARINO	ADMINISTRATION SYSTEMS	17
JOHNSON	ADMINISTRATION SYSTEMS	16
HENDERSON	OPERATIONS	16
SCHNEIDER	OPERATIONS	17

Note: The above query is different from the previous correlated subqueries in that the first subquery may return no values. Suppose that a department with three employees is being evaluated.

Working from bottom to top, the following occurs:



The inner-most subquery evaluates to 3. Thus, the expression “AND 5 <= 3” is false. Because that expression is false, no rows satisfy the search condition of the next subquery, and a null value is returned to the outer-most query. This causes the predicate “EDLEVEL > subquery)” to evaluate to the unknown truth value. The join condition “Y.WORKDEPT = DEPARTMENT.DEPTNO”, however, is always true:



The following figure is the “AND” truth table for search conditions; “TRUE AND UNKNOWN” causes the search condition in the query to be “UNKNOWN,” as indicated above.

	AND	T	F	?
T		T	F	?
F		F	F	F
?		?	F	?

No rows satisfy the search condition, so no employee is listed for department A00; exactly the result wanted in this case.

Using a Subquery to Test for the Existence of a Row

You can use a subquery to test for the *existence* of a row satisfying some condition. In this case, the subquery is linked to the outer-level query by the predicate EXISTS or NOT EXISTS. (Refer to the *DB2 Server for VSE & VM SQL Reference* manual for the syntax of the EXISTS predicate.)

When you link a subquery to an outer query by an EXISTS predicate, the subquery does not return a value. Rather, the EXISTS predicate is true if the answer set of the subquery contains one or more rows, and false if it contains no rows.

The EXISTS predicate is often used with correlated subqueries. The example below lists the departments that currently have no entries in the PROJECT table:

```

DECLARE C1 CURSOR FOR
SELECT  DEPTNO, DEPTNAME
FROM    DEPARTMENT X
WHERE   NOT EXISTS
        (SELECT *
         FROM PROJECT
          WHERE DEPTNO = X.DEPTNO)
ORDER BY DEPTNO

```

You may connect the EXISTS and NOT EXISTS predicates to other predicates by using AND and OR in the WHERE clause of the outer-level query.

Table Designation Rule for Correlated Subqueries

Unqualified correlated references are allowed. For example, assume that table EMP has a column named SALARY and that table DEPT has a column named BUDGET, but no column named SALARY.

```

SELECT * FROM EMP
WHERE EXISTS (SELECT * FROM DEPT
             WHERE BUDGET < SALARY)

```

In this example, the system checks the innermost FROM clause for a SALARY column. Not finding one, it then checks the next innermost FROM clause (which in this case is the outer FROM clause). It is only necessary to use a qualified correlated reference when you want the system to ignore a column with the same name in the innermost tables.

To assist you in these situations, a warning message SQLCODE +12 (SQLSTATE '01545') is issued whenever an SQL statement is executed that contains an unqualified correlated reference in a subquery.

Combining Queries into a Single Query: UNION

The UNION operator enables you to combine two or more outer-level queries into a single query. Each of the queries connected by UNION is executed to produce an answer set; these answer sets are then combined, and duplicate rows are eliminated from the result.

When ALL is used with UNION (that is, UNION ALL), duplicate rows are not eliminated when two or more outer-level queries are combined into a single query. If you are using the ORDER BY clause, you must write it after the last query in the UNION. The system applies the ordering to the combined answer set before it delivers the results to your program using the usual cursor mechanism.

It is possible (though unusual) to write a query using the UNION operator that does not return results with a cursor. In this instance, only one row must be retrieved from the tables, and an INTO clause must be placed only in the first query.

The UNION operator is useful when you want to merge lists of values derived from two or more tables and eliminate any duplicates from the final result. UNION ALL will give better performance, however, because no internal sort is done. This sort is done with the UNION operator to facilitate the elimination of duplicates.

When both UNION and UNION ALL are used in the same query, processing is from left-to-right. If the last union operation is UNION, the duplicates will be eliminated from the final results; if it is UNION ALL, the duplicates will not be eliminated. However, the left-to-right priority can be altered by the use of parenthesis. A parenthesized subselect is evaluated first, followed, from left-to-right, by the other components of the statement. For example, the results of the following two queries, where A, B, and C are subselects, could be quite different:

```
A UNION (B UNION ALL C)
(A UNION B) UNION ALL C
```

In the following example, the query returns all projects for which the estimated mean number of employees is greater than 0.50, and it returns all the projects where the proportion of employee time spent on the project is greater than 0.50:

```

SELECT PROJNO, 'MEAN'
FROM PROJ_ACT
WHERE ACSTAFF > .50

UNION

SELECT PROJNO, 'PROPORTION'
FROM EMP_ACT
WHERE EMPTIME > .50

```

The database manager combines the results of both queries, eliminates the duplicates, and returns the final result in ascending order.

Note: The ascending order is a direct result of the internal sort, which is performed to facilitate the elimination of duplicates.

```

PROJNO EXPRESSION
-----
AD3110 MEAN
AD3110 PROPORTION
AD3111 MEAN
AD3111 PROPORTION
AD3112 MEAN
AD3112 PROPORTION
AD3113 MEAN
AD3113 PROPORTION
IF1000 MEAN
IF1000 PROPORTION
IF2000 MEAN
IF2000 PROPORTION
MA2100 MEAN
MA2100 PROPORTION
MA2110 MEAN
MA2110 PROPORTION
MA2111 MEAN
MA2111 PROPORTION
MA2112 MEAN
MA2112 PROPORTION
MA2113 MEAN
MA2113 PROPORTION
OP1010 MEAN
OP1010 PROPORTION
OP2000 MEAN
OP2010 MEAN
OP2010 PROPORTION
OP2011 MEAN
OP2011 PROPORTION
OP2012 MEAN
OP2012 PROPORTION
PL2100 MEAN
PL2100 PROPORTION

```

To connect queries by the UNION operator, you must ensure that they obey the following rules:

- All corresponding items in the *select-lists* of the queries in the union must be *compatible*.
- An ORDER BY clause, if used, must be placed after the last query in the union. The order-list must contain only integers, not column names. In the example query above, ORDER BY 1 is acceptable, but ORDER BY PROJNO is not.
- None of the queries in a union may select long strings.
- A union may not be specified inside a subquery.
- A union may not be used in the definition of a view.
- VARCHAR and VARGRAPHIC values that differ only by trailing blanks are considered equal. One of the values will be eliminated as a duplicate value unless UNION ALL is selected.

Unions between columns that have the same data type and the same length produce a column with that type and length. If they are not of the same type and length but they are union-compatible, the resulting column-type is a combination of the two original columns.

The results of a UNION between two union-compatible items is summarized below. The first row and first column of the table represent the data-type of the first and second columns of the UNION join.

String Columns

	CHAR	VARCHAR	GRAPHIC	VARGRAPHIC
CHAR	CHAR	VARCHAR	ERROR	ERROR
VARCHAR	VARCHAR	VARCHAR	ERROR	ERROR
GRAPHIC	ERROR	ERROR	GRAPHIC	VARGRAPHIC
VARGRAPHIC	ERROR	ERROR	VARGRAPHIC	VARGRAPHIC

The length attribute of the resulting column will be the greater of the length attributes of the original columns.

The UNION operators between columns that have the same character subtype and CCSID produce a column with that subtype and CCSID. If they do not have the same subtype and CCSID, the resulting subtype and CCSID are determined following specific rules. For a detailed discussion of these rules, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

Numeric Columns

	SMALLINT	INTEGER	DECIMAL	SINGLE PRECISION	DOUBLE PRECISION
SMALLINT	SMALLINT	INTEGER	DECIMAL	DOUBLE PRECISION	DOUBLE PRECISION
INTEGER	INTEGER	INTEGER	DECIMAL	DOUBLE PRECISION	DOUBLE PRECISION
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DOUBLE PRECISION	DOUBLE PRECISION
SINGLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	SINGLE PRECISION	DOUBLE PRECISION
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION

When both of the original columns are DECIMAL data-types, special rules apply for determining the scale and precision of the resulting column.

Where s is the scale of the first column of the UNION join, s' is the scale of the second column, p is the precision of the first column, and p' is the precision of the second, the resulting column's precision is:

$$\text{MIN}(31, \text{MAX}(s , s') + \text{MAX}(p-s , p'-s'))$$

The scale of the resulting column is the maximum scale of the original columns of the UNION join, $\text{MAX}(s, s')$.

When a UNION is performed on a DECIMAL and either an INTEGER or SMALLINT column, the resulting column's scale and precision can be calculated with the previous formulas. However, remember to substitute 11 and 0 for the precision and scale of an INTEGER column, and 5 and 0 for a SMALLINT column.

Datetime/Timestamp Columns

	DATE	TIME	TIMESTAMP
DATE	DATE	ERROR	ERROR
TIME	ERROR	TIME	ERROR
TIMESTAMP	ERROR	ERROR	TIMESTAMP

Note: CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are not union-compatible with DATE, TIME, or TIMESTAMP.

SQL Comments within Static SQL Statements

You can use a comment as a separator within static SQL statements written in the various host languages. This comment is referred to as an SQL comment (as opposed to host language comments), and is identified by two consecutive hyphens (--) on the same line, not separated by a space and not part of a literal, a string of DBCS characters, a quoted identifier, or an embedded host language comment. In COBOL, the two hyphens must be preceded by a blank. The comment ends at the end of the line.

Here is the sample query from the previous discussion on UNION, documented with a few SQL comments:


```

SELECT PROJNO, 'MEAN'
FROM PROJ_ACT -- PROJECT ACTIVITY TABLE
WHERE ACSTAFF > .50
-- FIRST QUERY IS FOR ESTIMATED MEAN NUMBER OF EMPLOYEES
UNION
-- SECOND QUERY IS FOR PROPORTION OF EMPLOYEE TIME
SELECT PROJNO, 'PROPORTION'
FROM EMP_ACT -- EMPLOYEE ACTIVITY TABLE
WHERE EMPTIME > .50

```

The *DB2 Server for VSE & VM SQL Reference* manual for the detailed syntax rules on the use of SQL comments within application programs.

Using Stored Procedures

A stored procedure is a user-written application program that is compiled and stored at the server. When the database manager is running in multiple user mode, local applications or remote DRDA applications can invoke the stored procedure. Since the SQL statements issued by a stored procedure are local to the server, they do not incur the high network costs of distributed statements. Instead, a single network send and receive operation is used to invoke a series of SQL statements contained in the stored procedure.

Figure 29 and Figure 30 on page 96 illustrate how the use of stored procedures reduces network traffic by decreasing the number of commands that flow between the application requester and the application server.

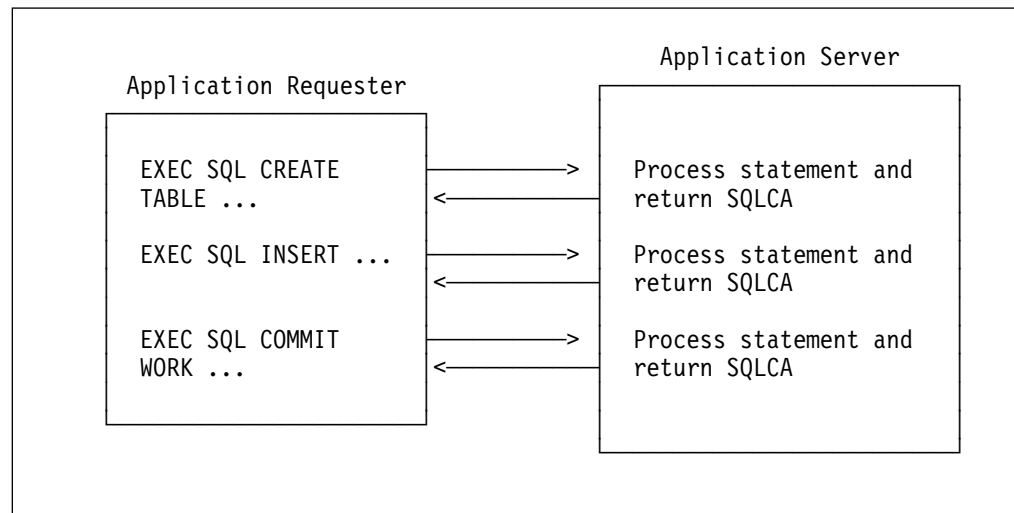


Figure 29. Without Stored Procedures

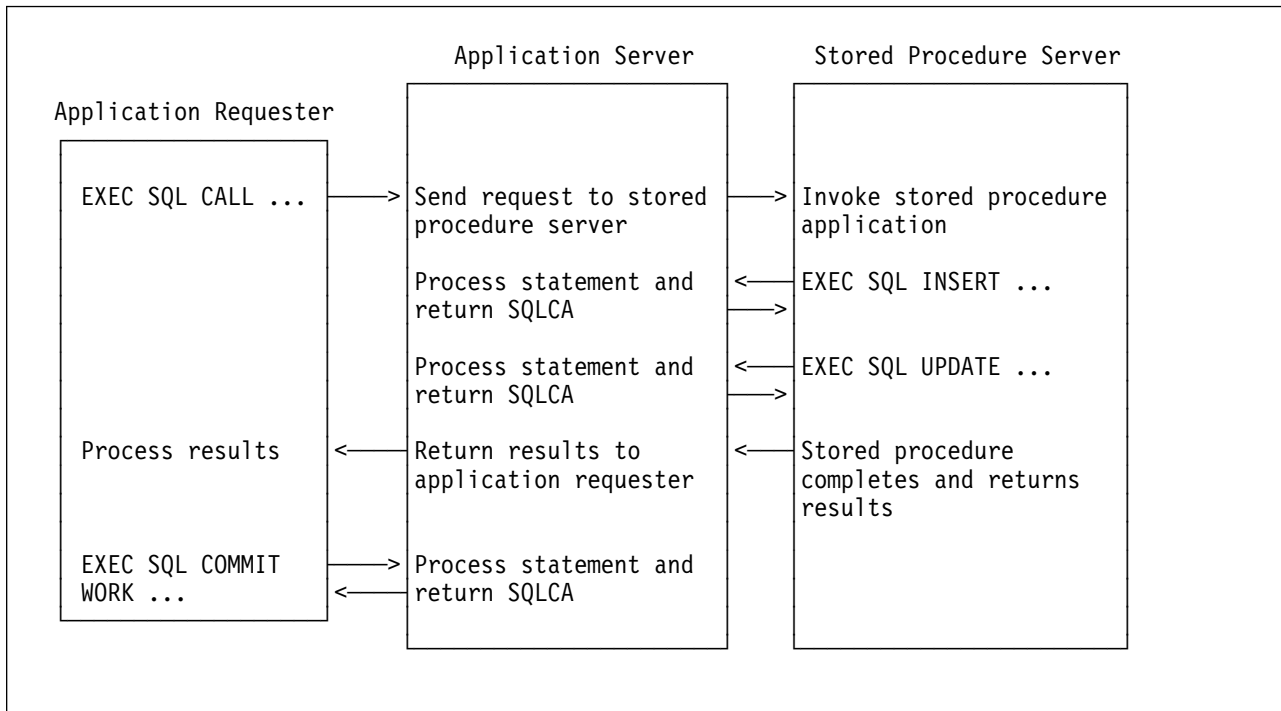


Figure 30. With Stored Procedures

For information on the stored procedure environment, including stored procedure servers, refer to the *DB2 Server for VSE Database Administration* manual.

There are several other benefits that can be gained through the use of stored procedures, including:

- In many applications, the integrity of the host variables used in SQL statements is critical to the business function provided by the application. For example, a debit/credit application might need to guarantee that the host variable values do not change between debit and credit operations. In these applications, the application designer would like to guarantee that sophisticated users cannot employ online debugging tools to manipulate the content of SQL statements or host variables used by the SQL application. By using stored procedures, the application designer can encapsulate the application's SQL statements into a single message to the server, which moves the sensitive processing beyond the reach of even the most sophisticated workstation user.
- Stored procedures can be used to hide the details of the database design from client applications. In addition to simplifying the writing of client applications, this means that if the database design is changed, only the stored procedure needs to be modified. The more client applications that use the stored procedure, the greater the benefit.
- Stored procedures can be used to hide sensitive data from application programs.
- Business logic can be encapsulated at the server, rather than being included in numerous application programs.
- It is easier to maintain an environment in which applications are kept at the server rather than spread across a number of requesters.

Writing Stored Procedures

Stored procedure that are to be used on a DB2 Server for VSE & VM database can be written in PL/I, COBOL, C, or Assembler. Stored procedures are very much like regular application programs, with the following exceptions:

- They must be LE compliant
- They cannot contain the following SQL statements: CONNECT, COMMIT, ROLLBACK, or CALL

Note: Stored procedures must be written as MAIN programs; they cannot be SUB programs.

The following is an example of a simple stored procedure. It contains one SQL statement that SELECTs the salary of a given employee from the SQLDBA.EMPLOYEE table. The employee number is provided as input, and the salary and the SQLCODE for the SELECT statement are returned.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. SAMP1.  
  
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
  
DATA DIVISION.  
FILE SECTION.  
WORKING-STORAGE SECTION.  
  
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
  
    01 CHAR6HV                PIC X(6).  
    01 SALHV                  PIC S9(7)V9(2) COMPUTATIONAL-3.  
  
    EXEC SQL END DECLARE SECTION END-EXEC.  
  
    EXEC SQL INCLUDE SQLCA END-EXEC.  
  
LINKAGE SECTION.  
    01 CHAR6                  PIC X(6).  
    01 SALARY                  PIC S9(7)V9(2) COMPUTATIONAL-3.  
    01 SQLCD                   PIC S9(9) COMP.  
  
PROCEDURE DIVISION USING CHAR6 SALARY SQLCD.  
  
* TURN OFF SQL EXCEPTION PROCESSING *  
    EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.  
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.  
    EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.  
  
    MOVE CHAR6 TO CHAR6HV.  
  
    EXEC SQL  
        SELECT SALARY INTO :SALHV FROM SQLDBA.EMPLOYEE  
            WHERE EMPNO = :CHAR6HV  
    END-EXEC.  
  
    MOVE SALHV TO SALARY.  
    MOVE SQLCODE TO SQLCD.  
  
    STOP RUN.
```

The following is an example of a CALL statement that could be used to invoke the procedure shown above:

```
CALL SAMP_PROC ('000250', :SALARY, :SQLCD)
```

The SQL CALL statement is discussed in more detail in “Calling Stored Procedures” on page 99.

Returning Information from the SQLCA

Information about the execution of SQL statements within a stored procedure is not returned to the application that invoked the stored procedure. If SQLCODE, SQLSTATE, or any other information from the SQLCA is required by the calling application, that information must be included in the parameter list of the stored procedure and the parameters must be set explicitly in the stored procedure. This is because there are many situations in which a negative SQLCODE does not necessarily indicate a problem (such as dropping a table that does not exist). The person who writes the stored procedure application must determine what SQLCODEs should be returned to the caller.

See “Writing Stored Procedures” on page 97 for an example of a stored procedure that returns an SQLCODE.

Language Environment® (LE) Considerations

As mentioned previously, stored procedures must be LE-compliant. IBM Language Environment for VSE/ESA establishes a common run-time environment for different programming languages. It combines essential run-time services, such as condition handling and storage management. All of these services are available through a set of interfaces that are consistent across programming languages. With LE, you can use one run-time environment for your applications, regardless of the application's programming languages or system resource requirements.

Language Environment is the prerequisite run-time environment for applications generated with the following IBM compiler products:

- IBM C for VSE/ESA
- IBM COBOL for VSE/ESA
- IBM PL/I for VSE/ESA

Stored procedures can be written in assembly language as long as the assembly language program uses the required macros to operate as an IBM Language Environment application program.

For complete details, see the Language Environment documentation.

Preparing to Run a Stored Procedure

Once the stored procedure has been written, it must be preprocessed, compiled, and linked like any application program, and the phase must be put in a library that is in the stored procedure server's search path. In addition, the CREATE PROCEDURE statement must be used to define the stored procedure to the database manager. See the *DB2 Server for VSE & VM SQL Reference* manual for information on the CREATE PROCEDURE statement.

Calling Stored Procedures

Once a stored procedure has been created and the CREATE PROCEDURE statement has been used to define it, it can be invoked. The SQL CALL statement is used in an application program to invoke a stored procedure. The syntax of the CALL statement is shown in Figure 31.

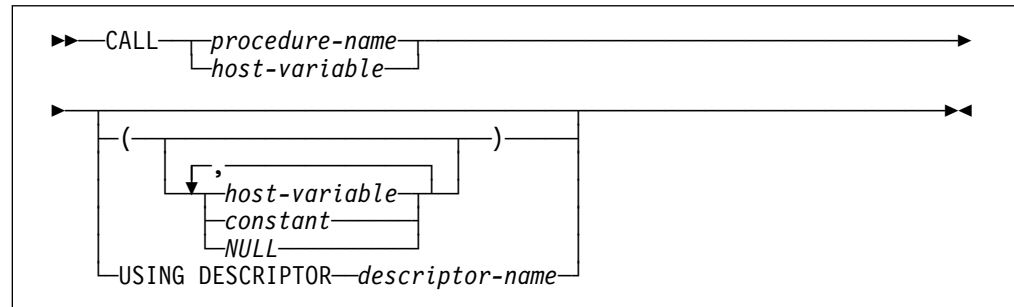


Figure 31. Syntax of SQL CALL statement

For a complete description of the CALL statement, see the *DB2 Server for VSE & VM SQL Reference* manual.

As indicated in Figure 31, the procedure name can be a host variable or a constant, and parameters can be provided in a parameter list or in a descriptor (SQLDA). A simple example of a CALL statement might look like this:

```
EXEC SQL CALL PROC1 ('000250', :lastname, :salary, :sqlcd)
```

The CALL statement shown above assumes that none of the input parameters can have null values. If you need to allow for null values, use indicator variables with the host variables, as follows:

```
EXEC SQL CALL PROC1 (:empno :empnoi,
                    :lastname :lnamei,
                    :salary :salaryi,
                    :sqlcd :sqlcdi)
```

If you do not know the parameter structure of the procedure, or if you prefer to use one structure rather than several host variables, you would use the following form of the CALL statement:

```
EXEC SQL CALL PROC1 USING DESCRIPTOR :sqlda
```

where *sqlda* is the name of an SQLDA. The parameter information must be put in the SQLDA before the CALL is issued.

The final example provides maximum flexibility:

```
EXEC SQL CALL :procname USING DESCRIPTOR :sqlda
```

where *sqlda* is the name of an SQLDA. The parameter information must be put in the SQLDA before the CALL is issued.

Authorization

Authorization for stored procedures is done on a package level. That is, the issuer of the CALL statement must be authorized to run the package associated with the stored procedure. See Chapter 9, “Assigning Authority and Privileges” on page 213 for more information on authorization.

AUTHIDs

On the CREATE PROCEDURE statement, you can specify an AUTHID. If you do, then only a user with that AUTHID can run the stored procedure. The AUTHID corresponds to the SQL ID of a connected user. This facility is useful for testing modifications to a stored procedure. It allows the database administrator to create a private copy of the stored procedure, modify and test it, without affecting the copy of the stored procedure that is publicly accessible. Once the stored procedure is fully tested, it can replace the existing, publicly accessible stored procedure.

Stored Procedure Parameters

The parameters for a stored procedure are defined on the CREATE PROCEDURE statement. The CREATE PROCEDURE statement makes an entry in SYSTEM.SYSPARMS for each parameter. The entry in SYSTEM.SYSPARMS indicates the datatype, size, and purpose (input, output, or both) of the parameter.

The stored procedure must have a declaration for each parameter that is passed to it. The declaration of each parameter must be compatible with the datatype and size specified for it in SYSTEM.SYSPARMS. Figure 32 shows the compatible definitions for parameters in C, COBOL, PL/I, and Assembler.

Figure 32. Definitions of Stored Procedure Parameters

SYSPARMS	C	COBOL	PL/I	Assembler
CHAR(n)	char varname[n+1]	PIC X(n)	CHAR(n)	CLn
CHAR(1)	char	PIC X(1)	CHAR(1)	CL1
VARCHAR(n)	char varname[n+1]	01 parm 49 parm1 PIC S9(4) COMP 49 parmd PIC X(n)	CHAR(n) VARYING	H,CLn
SMALLINT	short	PIC S9(4) COMP	BIN FIXED(15)	H
INTEGER	long	PIC S9(9) COMP	BIN FIXED(31)	F
DECIMAL(x,y)	DECIMAL[(p,[s])] or DEC[(p,[s])]	PIC S9(x-y)V9(y) COMP-3	DEC FIXED(x,y)	PLn['decimal constant'] or P'decimal constant'
REAL	float	COMP-1	BIN FLOAT(21)	E
FLOAT	double	COMP-2	BIN FLOAT(53)	D
GRAPHIC(n)	not supported	PIC G(n) DISPLAY-1 or PIC N(n)	GRAPHIC(n)	not supported
VARGRAPHIC(n)	not supported	01 parm 49 parm1 PIC S9(4) COMP 49 parmd PIC G(n) USAGE IS DISPLAY-1 or 49 parmd PIC N(n)	GRAPHIC(n) VARYING	not supported

Each of the high-level language definitions for stored procedure parameters support only a single instance (scalar value) of the parameter. There is no support for

structure, array, or vector parameters. In some applications, it may be necessary to return a table of results, where the table represents multiple occurrences of one or more of the parameters passed to the stored procedure. Since this support is not provided by the SQL CALL statement, one of the following techniques may be used by the application to provide the required capability:

- If the data to be returned is in a table in the database, the calling program can fetch the rows directly using SQL. Since a DRDA requester can intermix SELECT and CALL statements in a unit of work, the DRDA block fetch protocol can be used to retrieve the required data efficiently.
- Tabular data can be converted to string format and returned as a character string parameter to the calling program. The calling program and the stored procedure can establish a convention for interpreting the content of the character string. For example, the SQL CALL statement can pass a 1920 byte character string parameter to a stored procedure, allowing the stored procedure to return a 24 by 80 screen image to the calling program.

Datatype Compatibility

The datatype of a parameter provided on the CALL does not have to be identical to the datatype expected by the stored procedure, but it must be compatible. That is, if the stored procedure expects a CHAR(4) parameter, the caller can provide a character or varchar value with a length of 4 or less. Similarly, if the procedure expects an integer, most numeric datatypes (decimal, smallint, float) are acceptable, as long as the number is not too large to be represented by an integer. In general, datatypes that are considered compatible in other SQL statements are also considered compatible in an SQL CALL. That is, if the value being provided on the SQL CALL could be inserted into a column that has the same datatype as the stored procedure parameter, then it is valid for the SQL CALL statement.

For more information on datatype compatibility, see the *DB2 Server for VSE & VM SQL Reference* manual.

Conventions for Passing Stored Procedure Parameters

When an SQL CALL statement is issued, DB2 Server for VSE & VM builds a parameter list for the stored procedure, containing the parameters provided on the SQL CALL statement. When the initial parameter list is built, the parameters contain the values established on entry to the SQL CALL statement. Eventually, the database manager will run the stored procedure and return values for the parameters to the calling program. If a stored procedure fails to set one or more of the output parameters, the database manager will not detect this fact. Instead, it will return the output parameter(s) to the calling program, with the value(s) established on entry to the SQL CALL statement.

In order for the stored procedure to receive parameters correctly, the stored procedure must be coded to accept the parameter list supplied by the database manager. DB2 Server for VSE & VM supports two parameter list conventions. The parameter list convention is determined by the value of the PARAMETERSTYLE column in the SYSTEM.SYSROUTINES catalog table, which can be GENERAL or GENERAL WITH NULLS.

The GENERAL Linkage Convention

If the GENERAL linkage convention is used:

- Input parameters cannot be NULL.
- NULLs can be passed for output parameters only.
- The stored procedure cannot return NULLs for output parameters.
- A parameter must be defined in the stored procedure for each parameter passed in the SQL CALL statement.

For performance reasons, the calling application may choose to pass null indicators with the output parameters on the SQL CALL statement. If the null indicator associated with an output parameter is negative on entry to the SQL CALL statement, the application requester transmits only the null indicator to the server. This can be beneficial when dealing with large output parameters, since the entire output parameter is not transmitted to the server. Upon successful completion of the SQL CALL statement, none of the null indicators associated with the output parameters will be null, since the stored procedure is restricted to non-null parameter values.

When the GENERAL parameter list format is used, register 1 points to a list of addresses, which in turn point to the individual parameters. Figure 33 describes the GENERAL parameter list convention.

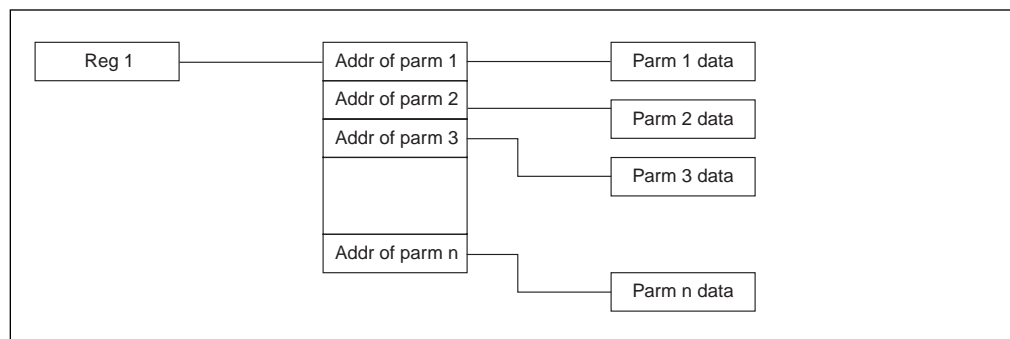


Figure 33. GENERAL parameter list

The GENERAL WITH NULLS Linkage Convention

This is the default. If the GENERAL WITH NULLS linkage convention is used:

- Input parameters can be NULL. This is achieved through the use of indicator variables, or by specifying the keyword NULL.
- The stored procedure can return NULLs for output parameters, by using indicator variables.
- A parameter must be defined in the stored procedure for each parameter passed in the SQL CALL statement. An array of indicator variables, with one indicator variable for each parameter, must also be defined in the stored procedure.

The indicator variables are passed to the stored procedure as a single parameter - an array of SMALLINT variables with an element for each indicator variable.

Figure 34 on page 103 describes the GENERAL WITH NULLS parameter list convention.

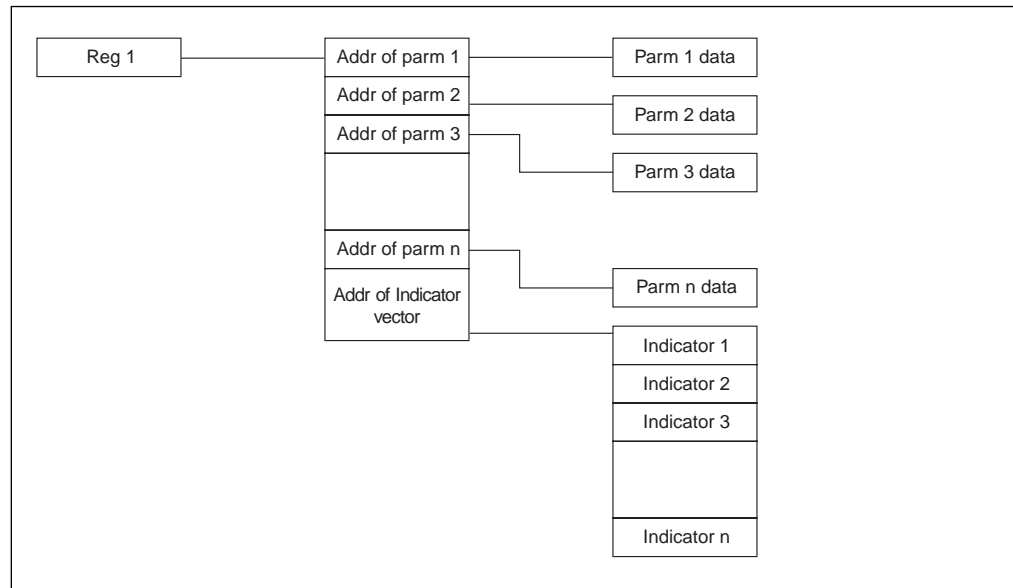


Figure 34. GENERAL WITH NULLS parameter list

The stored procedure must determine which input parameters are null by examining the array of indicator variables. The stored procedure must also assign values to the indicator variables when returning the output parameters to the calling program.

The array of indicator variables is not defined in the PARMLIST column of SYSTEM.SYSROUTINES, and is not specified as a parameter in the SQL CALL statement. In the SQL CALL statement in the client program, the indicator variables are coded after each parameter, for example:

```
EXEC SQL CALL PROCX (:parm1:indicator1, :parm2:indicator2)
or
EXEC SQL CALL PROCX (:parm1 INDICATOR :indicator1, :parm2 INDICATOR :indicator2)
```

In order to support the linkage conventions described above, the high level language application must be coded to support the required parameter list convention.

Coding Examples

For examples of how to code stored procedures to receive and return parameters in C, COBOL, PL/I, or Assembler, refer to the appendix for that language.

Special Considerations for C

The PLIST(OS) run-time option must be supplied.

Special Considerations for PL/I

The NOEXECOPS procedure option must be supplied.

Result Sets

In addition to returning parameters, a stored procedure can return query data, known as result sets. A result set is defined by declaring a cursor with the WITH RETURN clause, opening the cursor within the stored procedure, and leaving it open when the procedure returns. The resulting rows of data that can be fetched constitute a result set.

Notes:

1. For a procedure to return result sets, the RESULT_SETS column in the SYSTEM.SYSROUTINES entry for that procedure must contain a non-zero value.
2. The DB2 Server for VSE & VM requester does not have the capability to process result sets. As a result, DB2 Server for VSE & VM returns result sets only to DRDA clients other than DB2 Server for VSE & VM (for example, DB2 Universal Database).
3. A DB2 Server for VSE & VM application requester can issue the CALL statement in SQLDS protocol only.
4. If any FETCHes are issued within the stored procedure, the result set rows returned to the client start with the row after the last row that was fetched within the stored procedure. That is, if the stored procedure issues three FETCHes, the result set returned to the client starts with the fourth row.
5. The stored procedure must not use blocking. This is because if blocking is on, the application server returns a full block of rows when a FETCH is issued, leaving the cursor positioned on the row after the last row of the block. If the stored procedure does not FETCH all of the rows in the block, the rows that have already been returned to the stored procedure will not be returned to the application requester.

For information on how to process result sets on the client side, refer to the following manuals:

1. *IBM DB2 Universal Database Call Level Interface Guide and Reference*
2. *DB2 for OS/390 Application Programming and SQL Guide*.

Chapter 4. Preprocessing and Running a Program

Contents

Defining the Steps to Execute the Program	106
Using 31-Bit Addressing	106
How DB2 Establishes User IDs for CICS/VSE Transactions	107
User IDs for Remote CICS/VSE Transactions	108
Preprocessing the Program	108
Preprocessing by Mode	109
Using Multiple User Mode	110
Using Single User Mode	111
Defining the Preprocessing Parameters	112
Using the Preprocessor Option Member	121
Using the Flagger at Preprocessor Time	122
Using the CICS/VSE Translator	123
Improving Performance Using Preprocessing Parameters	124
Selecting the Isolation Level to Lock Data	124
Using the Blocking Option to Process Rows in Groups	128
Imposing Blocking Restrictions	130
Using the Blocking Option in DRDA Protocol	130
Using the INCLUDE Statement	131
Including External Source Members	131
Including a Library Source	131
Compiling the Program	132
Link-Editing and Loading the Program	132
Link-Editing the Program with Supplementary Information	132
Including Relocatable Modules	132
Including CICS/VSE Procedures	133
Including Extra Linkage Modules	133
Running the Program	134
Using a Consistency Token	134
Loading the Package and Rebinding	134
Running by Mode	135
Using Multiple User Mode	135
Using Single User Mode	135
Running under CICS/VSE Support	136
Accessing Other DB2 Family Application Servers	136
Installing Applications that Access the Database Manager	136
Installing a Batch Application	136
Installing an Online CICS/VSE Application	137
Distributing Packages across Like and Unlike Systems	139
Creating a Package Using CBND	140

Defining the Steps to Execute the Program

After your program is coded, you must follow a series of steps to prepare it to be run. The number of steps varies depending on the host language of the program and the environment in which the program is running. However, the steps below are common in each case. In order to run your DB2 Server for VSE application program you must:

- Preprocess the SQL code
- Compile the program
- Link-Edit and Load the program
- Run the program.

If the program is an online program, the CICS statements have to be “translated” before the program is compiled.

When the database manager was installed, your installation may have optionally chosen to generate the starter database. The statements

```
// EXEC PROC=ARIS61DB  
// EXEC PROC=ARIS61PL
```

are contained in single user mode job control examples throughout this section in order to identify the database and the libraries used.

The ARIS61DB procedure contains the DLBL statements required for accessing the starter database. The ARIS61PL procedure identifies the DB2 Server for VSE production libraries. To access a different database, you must substitute a different procedure (or your own DLBL statements) for ARIS61DB. To access your own libraries, you must substitute a different procedure (or your own VSE LIBDEF statements) for ARIS61PL.

Determine if the DB2 Server for VSE database and library definition statements are required by contacting the person who installed the system. Or, refer to the *DB2 Server for VSE Installation* manual. This manual contains a description of the database and library definition job control statements required for a database, and the product-supplied procedures available.

Using 31-Bit Addressing

The application server runs in 31-bit addressing mode (AMODE 31). In single user mode, the database manager transfers control to the application in the addressing mode established during link editing of the program. The database manager continues to operate in 31-bit addressing mode when control returns from the application program.

If you are writing a multiple user mode application, your application is not affected by the addressing mode of the application server

Refer to the *DB2 Server for VSE System Administration* manual for information on single user mode and user exits.

How DB2 Establishes User IDs for CICS/VSE Transactions

Online DB2 Server for VSE transactions need not issue an SQL CONNECT to establish the user id within the database manager. When a transaction does not issue a CONNECT statement with the “*userid IDENTIFIED BY password*” clause, DB2 Server for VSE online support attempts to establish the user ID for the transaction.

This implicit CONNECT capability is useful if your installation requires its terminal users to sign-on using the CESN transaction. For many DB2 Server for VSE transactions, your installation might consider the sign-on verification sufficient. It may also be useful if your installation has just installed the system, and finds it convenient to have all users identified by one name (for example, TESTUSER).

DB2 Server for VSE online support establishes a user ID for CICS/VSE transactions as follows:

1. If the transaction issues an SQL CONNECT statement with the “*userid IDENTIFIED BY password*” clause, the user ID is established explicitly for the application.
2. If the transaction does not issue such a CONNECT statement, the online support attempts to establish the user ID implicitly as follows:
 - a. If the transaction had a user ID established for a previous logical unit of work, and the previous logical unit of work did not specify the RELEASE option for COMMIT WORK or ROLLBACK WORK, that user ID is used. (Remember that the connection to the application server is dropped every time a logical unit of work ends with release; thus, the user ID has to be re-established if the transaction has more than one logical unit of work ending with release.) Otherwise...
 - b. If the user has signed on using the CESN transaction and the transaction is associated with a terminal, the sign-on user ID is used for the user ID. Otherwise...
 - c. The user ID that was specified as an input parameter to the transaction that enabled the DB2 Server for VSE online support to a particular server becomes the user ID. This transaction could be either CIRB or CIRA. The person that invoked CIRB or CIRA will know what the user ID is. However...
 - d. A user ID need not be specified when CIRB or CIRA is invoked. It is an optional parameter. If no user ID was specified, and none of the previous cases applied, the user ID established for your transaction is CICSUSER.

Once the user ID is determined as described above, one more requirement must be met in order to successfully complete the connection to the application server. Either the specific user ID must have been granted CONNECT authority or “ALLUSERS” must have been granted CONNECT authority. “ALLUSERS” is a special user ID that permits any user ID to be implicitly connected without having been specifically granted CONNECT authority. “ALLUSERS” can be used by the DBA to turn on or off the implicit connect capability.

Earlier in this book, it was recommended that you always explicitly connect to the application server to avoid confusion. However, it is recognized that many installations have terminal operators that need not be aware of DB2 Server for VSE

user ID or authorization capabilities. In these cases, the DB2 Server for VSE implicit CONNECT support can be very useful. For more details on the CICS transaction environment, see the *DB2 Server for VSE Database Administration* manual.

User IDs for Remote CICS/VSE Transactions

For online DB2 Server for VSE transactions which are accessing a **remote** server and which issued an SQL CONNECT statement with the “*userid IDENTIFIED BY password*” clause to establish the user ID within the database manager, the user ID is established explicitly for the transaction.

For online DB2 Server for VSE transactions which are accessing a **remote** server and which did not issue an SQL CONNECT statement with the “*userid IDENTIFIED BY password*” clause to establish the user id within the database manager, the Online Resource Adapter will attempt to establish the user ID for the transaction implicitly as follows:

1. If the transaction had a user ID established for a previous remote logical unit of work, and the previous logical unit of work did not specify the RELEASE option for COMMIT WORK or ROLLBACK WORK, and the transaction did not switch to another application server, that user ID and its corresponding password are used. (Remember that every time a logical unit of work ends with RELEASE or the transaction switched to another application server, and you enter another SQL statement, you are implicitly connected as the CICS signon userid. Therefore, the user ID has to be re-established if the transaction has more than one logical unit of work ending with RELEASE or if the transaction is switching application servers.)
2. The user ID returned by the CICS ASSIGN command is used for the user ID.

Preprocessing the Program

Preprocessing performs the following actions :

- It changes the SQL source code so that it can be processed during host language compiling
- It does either or both of the following:
 - It converts the SQL statements into a package, and binds the package to the database.
 - It creates a bind file so that it can subsequently be used by the online utility (CBND) to create a package in a remote (or local) database.

The preprocessor replaces all the SQL statements in the program with host language code that invokes the new package. The new version of the program also contains the SQL statements in comment form. The package, created either by the preprocessor or by CBND, contains information to carry out the SQL requests made by the program. The database manager follows the best access path to the data for each SQL statement in the program, using available indexes and data statistics of which the system keeps track.

When the program is run, the new code calls the system to handle each SQL statement. It also links the program to the application server and translates messages and statements between the two.

The preprocessors supplied with the database manager have the following program names:

- ARIPRPA - Assembler Preprocessor
- ARIPRPB - C Preprocessor
- ARIPRPC - COBOL Preprocessor
- ARIPRPF - FORTRAN Preprocessor
- ARIPRPP - PL/I Preprocessor

You preprocess your program to prepare it to use the system. To preprocess your program, you invoke the appropriate preprocessor through VSE job control statements. A job control ASSGN SYSIPT statement should point to your source program. Other job control statements must point to where the preprocessor should place the modified source-program output (ASSGN SYSPCH) and printed output (ASSGN SYSLST).

You can suppress SYSPCH and SYSLST output through NOPRINT and NOPUNCH parameters to the preprocessor.

The preprocessor requires the job control statement DLBL SQLGLOB. In addition, when you want the preprocessor to generate a bind file, job control statements DLBL SQLBIND and DLBL BINDWKF are required. These DLBL statements must be provided in either the preprocess job control or the system standard label subarea. For more information see “Creating a Package Using CBND” on page 140.

You can supply preprocessor parameters through the VSE EXEC statement PARM keyword. The preprocessor parameters are described later in this chapter.

If the preprocessor encounters an SQL error, it inserts statements in the modified source code that cause a subsequent compile (or assemble) to fail.

If the preprocessor encounters a severe error in an SQL statement, all processing stops. Syntactic checking is only performed on subsequent SQL statements if the error is not severe. The preprocessor also puts statements in the preprocessed program which will cause a subsequent compile to fail. If successful, the preprocessor places an entry in the SYSTEM.SYSACCESS catalog to record the newly created package.

Different job control is required to invoke the preprocessors depending on whether the system is running in multiple user or single user mode.

Preprocessing by Mode

Prior to invoking any of the preprocessors, data sets for input, output, and work files must be assigned. Data sets used by each of the preprocessors are shown in Figure 35 and Figure 36 on page 110. Many VSE systems may already have the logical units assigned and data set labels defined during the IPL procedure.

Figure 35. Data Sets Required by All Preprocessors

Data Set	Used For
SYSIPT	Source input is read from here
SYSPCH	Modified source output is written here
SYSLST	Print (report) output is written here
SQLGLOB	CHARNAME and DBCS options are obtained from this VSAM file
SQLBIND	Bind files are written to this VSAM file (needed only if the bind preprocessor option is specified).
BINDWKF	A VSAM work file used when performing VSAM I/O against the bind file (needed only if the bind preprocessor option is specified).

Figure 36. Work Files Required for Each Preprocessor

Preprocessor	Data Set	Requirement
Assembler	SYS001	The file must be the same size as the source input file.
C	SYS001	The file must be the same size as the source input file.
COBOL	SYS001	One logical record for each line of source code plus about 10 records for each SQL statement. Also, add an allowance for diagnostic messages.
	SYS002	Approximately 20 to 60 logical records for each SQL statement. The number you should reserve depends on the complexity of the SQL statement — particularly the number of host variables referenced.
	SYS003	Approximately 10 to 40 logical records for each SQL statement. (This number also depends on host variable references.)
	SYS004	The file must be the same size as the source input file.
FORTRAN	SYS001	One logical record for each line of source code plus about 10 records for each SQL statement. Also, add an allowance for diagnostic messages.
	SYS002	Approximately 20 to 60 logical records for each SQL statement. The number you should reserve depends on the complexity of the SQL statement — particularly the number of host variables referenced.
	SYS003	The file must be the same size as the source input file.
PL/I	SYS001	The file must be the same size as the source input file.

Using Multiple User Mode

When invoking any preprocessor in multiple user mode, it is recommended that you specify SIZE=AUTO.

Figure 37 on page 111 shows generic job control that invokes a preprocessor in multiple user mode.


```

// JOB jobname
// ASSGN SYSIPT,cuu          *-- } Optional - may be
// ASSGN SYSPCH,cuu         *-- } assigned by
// ASSGN SYSLST,cuu        *-- } standard label
// ASSGN SYSxxx,cuu        *-- } Preprocessor work files
// DLBL SQLGLOB,.....,DISP=(OLD,KEEP) *-- } Optional - may be
// DLBL SQLBIND,.....,DISP=(OLD,KEEP) *-- } provided by
// DLBL BINDWKF,.....      *-- } standard label
*
// EXEC PGM=ARIPRPx,SIZE=AUTO,PARM='prepparms'
.
. Input card stream if SYSRDR and SYSIPT are
. assigned to the same device or file
.
/*
/&

```

Note:

1. JCL must be changed to specify the correct Device address and Library definitions
2. Replace ARIPRPx with preprocessor name
3. PrepParms are discussed in “Defining the Preprocessing Parameters” on page 112.
4. You can code your own LIBDEF statements.
Refer to the *DB2 Server for VSE Installation* manual for more information on coding LIBDEF statements.
5. You can replace the input card stream with the READ MEMBER statement if the source file has been cataloged as member type A.
Refer to the *DB2 Server for VSE Installation* manual for more information on the READ MEMBER statement.

Figure 37. Invoking the DB2 Server for VSE Preprocessor in Multiple User Mode

Using Single User Mode

When invoking any preprocessor in single user mode, you must specify SIZE=AUTO. The job control for single user mode initializes the system with the preprocessor name and desired parameters. Figure 38 shows generic job control that invokes a preprocessor in single user mode:

The job control starts the system and invokes the preprocessor. The EXEC statement initializes the system in single user mode and passes to it (as a parameter) the name of the preprocessor (ARIPRPx). Note that the slash (/) is actually written in the EXEC statement. It separates the general DB2 Server for VSE options (such as SYSMODE and PROGRAM) from the preprocessor options (such as PREPNAME and USERID). For *prepparms*, specify desired preprocessor parameters; the parameters are in the following section.

The *DB2 Server for VSE & VM Operation* manual lists all the DB2 Server for VSE initialization parameters for single user mode. However, the person who installs the system should determine what the best initialization parameters for your installation are, and pass these on to you.

Note: COBOL programs invoking COBOL SORT cannot be run in single user mode.

```

// JOB jobname
// EXEC PROC=ARIS61DB          *-- DB2 for VSE Starter database
// EXEC PROC=ARIS61PL          *-- Library definition
// ASSGN SYSIPT,cuu            *-- } Optional - may be
// ASSGN SYSPCH,cuu            *-- } assigned by
// ASSGN SYSLST,cuu            *-- } standard label
// ASSGN SYSxxx,cuu            *-- Preprocessor workfiles
// DLBL SQLGLOB,....,DISP=(OLD,KEEP) *-- } Optional - may be
// DLBL SQLBIND,....,DISP=(OLD,KEEP) *-- } provided by
// DLBL BINDWKF,....           *-- } standard label
*
// EXEC PGM=ARISQLDS,SIZE=AUTO,PARM='SYSMODE=S,LOGMODE=A,          *
        PROGRAM=ARIPRPx/prepparms'
.
. Input card stream if SYSRDR and SYSIPT are
. assigned to the same device or file
.
/*
/&

```

Note:

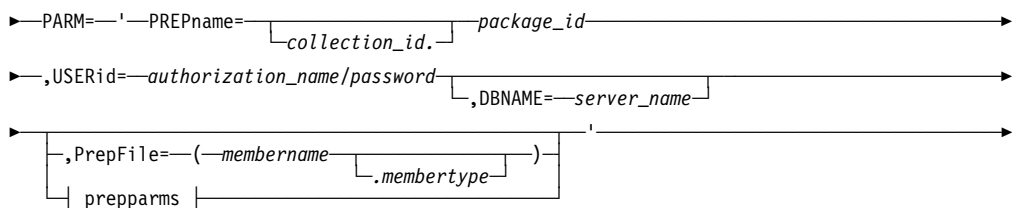
1. JCL must be changed to specify the correct Device address
2. Replace ARIPRPx with preprocessor name
3. PrepParms are discussed in "Defining the Preprocessing Parameters" on page 112.

Figure 38. Invoking the DB2 Server for VSE Preprocessor in Single User Mode

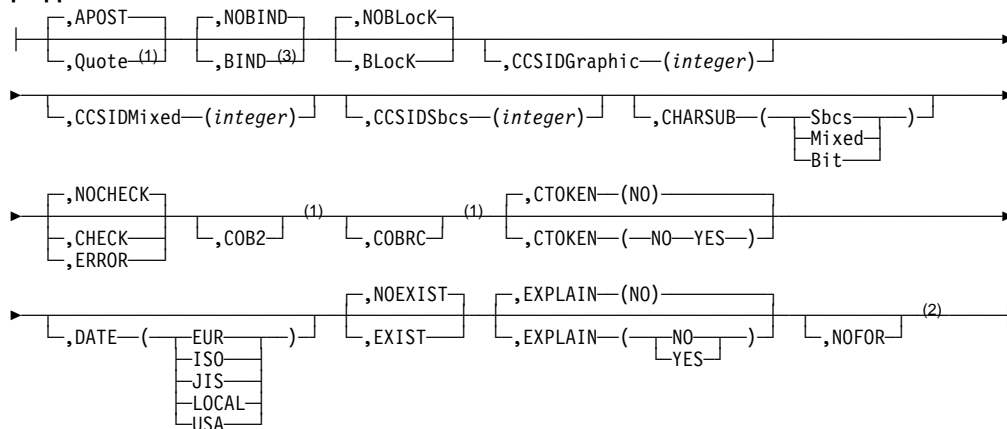
Defining the Preprocessing Parameters

The following are parameters for all DB2 Server for VSE preprocessors unless otherwise noted.

Program Preparation Parameters



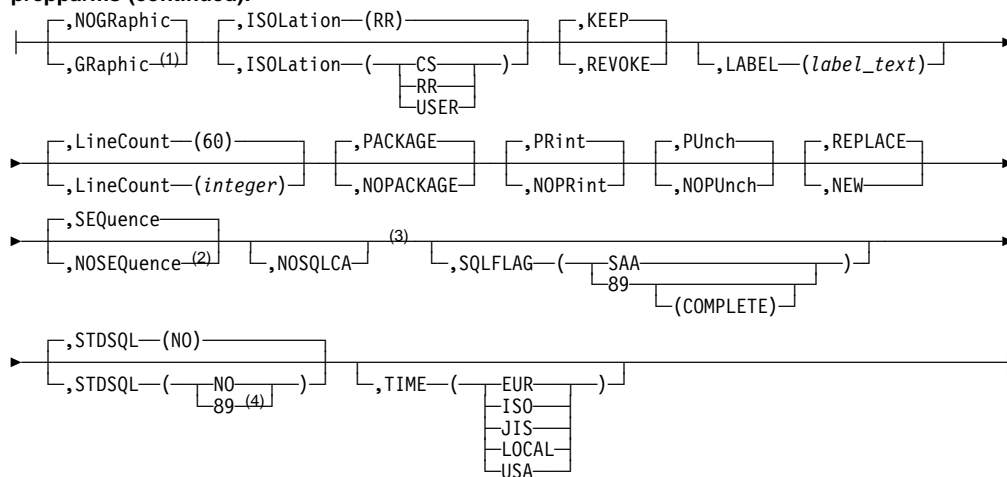
preparms:



Notes:

- 1 COBOL only.
- 2 Implied if STDSQL(89) is specified.
- 3 Ignored by Fortran.

preparms (continued):



Notes:

- 1 PL/1 and COBOL only.
- 2 C only.
- 3 Implied if STDSQL(89) is specified.
- 4 86 is a synonym for 89.

Specify these parameters using the PARM keyword of the VSE job control EXEC statement. The order in which you specify them is unimportant. You must separate all preprocessor parameters by a comma or by one or more blanks. (See Figure 39.)

Note: The maximum number of bytes that can be included within the quotation marks after the PARM keyword is 100. Therefore, you should take advantage of the abbreviations and defaults for the preprocessor parameters.

Figure 39. Specifying Preprocessor Parameters

Multiple User Mode

```
// EXEC PGM=ARIPRPx,SIZE=AUTO,PARM='PREP=MYJOB,USERID=SAM/SECRET'
```

Single User Mode

```

                                Col. 72 -----
// EXEC PGM=ARISQLDS,SIZE=AUTO,PARM='SYSMODE=S,LOGMODE=A,          |
                                *                                     |
                                PROGM=ARIPRPx/PREPNAME=MYJOB,USERID=SAM/SECRET'
```

In single user mode, the first / separates the preprocessor parameters from the DB2 Server for VSE initialization parameters. The second / separates the authorization-name from the password in the USERID preprocessor parameter (described below).

If you want to keep the authorization-name and password secret, you must suppress the printout of the job control EXEC statement that contains the preprocessor parameters. To do this, surround the preprocessor job control with the statements shown in Figure 40.

```

NOLOG          <--- Suppress display of JCL on the system
// OPTION NOLOG      <--- Suppress output to SYSLST
    (c)
    (c)
    (c)
LOG            <--- Resume output to the CONSOLE
// OPTION LOG       <--- Resume output to SYSLST
```

Figure 40. Suppressing Job Control Display of Authorization-Name and Password

If the program is preprocessed successfully and PACKAGE was specified, an entry is made in the DB2 Server for VSE catalog table SYSTEM.SYSACCESS. The CREATOR column is set to the value specified for authorization-name; the TNAME column is set to the value specified for PREPNAME. For more information about the DB2 Server for VSE catalog, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

PREPname=package_id

PREPname=collection_id.package-id

The *collection_id.package_id* is the name by which the database manager identifies the package. The *collection_id* portion is optional, and fully qualifies the *package_id* and any unqualified objects referenced within the package.

If *collection_id* is not specified, it defaults to the *authorization_name* specified on the USERid parameter. If it is specified, it must equal the *authorization_name* specified on the USERid parameter.

USERid=authorization_name/password

The *authorization_name* is the name by which the application server identifies the owner of a package. The password should agree with the one established for this *authorization_name* by a DB2 Server for VSE GRANT CONNECT statement. This information is used when executing a CONNECT statement to gain access to the application server, which determines whether proper authorization exists for the static SQL statements in the program.

DBNAME=(server_name)

This parameter identifies the name of the application server to be accessed by the SQL statements in the preprocessor source file. This parameter is used as the *server_name* in the TO clause of the CONNECT statement executed at preprocessing time. If this parameter is not specified, the preprocessor accesses the default application server.

Refer to the *DB2 Server for VSE System Administration* manual for a discussion of the default application server.

PrepFile=(membername)**PrepFile=(membername.membertype)**

The PrepFile parameter identifies the membername and optionally membertype of a VSE source member containing the list of preprocessor parameters. If *membertype* is not specified, it defaults to A.

The following parameters can be specified in the PrepFile or in the preprocessor parameters. For a more detailed discussion of the options file, see "Using the Preprocessor Option Member" on page 121.

PrepParm

The following parameters specify the preprocessor options.

APOST**Quote (COBOL preprocessor only)**

If you do not specify this parameter, the preprocessor defaults to APOST and generates single quotation mark (') delimiters for its internal source declarations. The Quote preprocessor parameter should be used whenever the Quote parameter is used in the COBOL compiler. Quote causes the preprocessor to use double quotation marks (") as constant delimiters in the VALUE clauses of the declarations it generates.

The use of a single quotation mark (') or double quotation marks (") in SQL statements is not affected by this parameter.

APOST/Quote is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

NOBIND**BIND**

If you specify the NOBIND parameter, the preprocessor does not create a bind file; NOBIND is the default.

If you specify the BIND parameter, the preprocessor creates a bind file. A bind file will not be created if NOCHECK is in effect and there was an error found during SQL statement validation. BIND is ignored if CHECK is specified. For a more detailed discussion of the bind file, see "Creating a Package Using CBND" on page 140.

Note: The Fortran preprocessor ignores the BIND parameter, if specified.

NOBlock**BLOCK**

When the BLOCK parameter is specified, all eligible query cursors return results in groups of rows, and all eligible insert cursors process inserts in groups of rows. If you do not specify this parameter, NOBlock is the default. This improves the performance of programs running in multiple user mode, where many rows are inserted or retrieved. For a discussion of

eligible cursors, see “Using the Blocking Option to Process Rows in Groups” on page 128.

When NOBLOCK is specified, rows are not grouped.

BLOCK/NOBLOCK is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

If you want to change the BLOCK option, you must recompile (or reassemble) and relink your program after preprocessing it. You must also use CBND to rebuild the package if BIND is specified. Preprocessing alone does not change the BLOCK setting. You must also use CBND to rebuild the package if BIND is specified.

CCSIDGraphic (*integer*)

This parameter specifies the default CCSID attribute to be used for graphic columns created in the package, if an explicit CCSID is not specified on the CREATE or ALTER statements in the package. If this parameter is not specified, the target application server uses the system default. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

CCSIDMixed (*integer*)

This parameter specifies the default CCSID attribute to be used for character columns created with the mixed subtype in the package, if an explicit CCSID is not specified on the CREATE or ALTER statements in the package. If this parameter is not specified, the target application server uses the system default. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

CCSIDSbcs (*integer*)

This parameter specifies the default CCSID attribute to be used for character columns created with the SBCS subtype in the package, if an explicit CCSID is not specified on the CREATE or ALTER statements in the package. If this parameter is not specified, the target application server uses the system default. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

CHARSUB (Sbcs)

CHARSUB (Mixed)

CHARSUB (Bit)

This parameter specifies the character subtype attribute to be used for character columns created in the package, if an explicit subtype or CCSID is not specified. If you do not specify this parameter, the target application server uses the system default. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

NOCHECK

CHECK

ERROR

If you specify the NOCHECK parameter, the preprocessor executes normally; that is, it validates all SQL statements when performing package functions. If NOPACKAGE is specified, package functions are not performed and so NOCHECK is ignored in this case. NOCHECK will be stored in the BIND file header if BIND is specified and a bind file is successfully created after preprocessing; NOCHECK is the default.

If you specify the CHECK parameter, the preprocessor checks all SQL statements for validity and generates error messages if necessary, but does not generate a package or bind file. PACKAGE and BIND are ignored if CHECK is specified.

If you specify ERROR, the preprocessor executes normally except that most statement-parsing errors are tolerated. When one of these errors is detected, the preprocessor generates an error message in the output listing and the modified source code in commented form, and continues processing. The program can be compiled and executed, but the erroneous statement cannot be executed. If NOPACKAGE is specified, package functions are not performed and so ERROR is ignored in this case. ERROR will be stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

You should use the ERROR option when you are also generating a bind file and intend to bind it against a remote application server, where at least one statement in the program is specific to an unlike application server.

COB2 (COBOL preprocessor only)

This parameter enables you to use certain COBOL II functions that are supported by the COBOL II Release 3 compiler and later. Refer to "Using the COB2 Parameter" on page 300 for a list of those functions.

COBRC (COBOL preprocessor only)

If this parameter is specified, the preprocessor will generate the statement 'MOVE ZEROS TO RETURN-CODE' after it generates a call to ARIPRDI. For more information, see "Using the COBRC Parameter" on page 300..

CTOKEN (NO)

CTOKEN (YES)

This parameter causes the preprocessor to store a consistency token in the modified source code and the package. At run time, consistency tokens in the program's load module and package must match before the application server executes the package. CTOKEN(NO) is the default. If CTOKEN(YES) is specified, the consistency token generated by the preprocessor will be an 8-byte 390 Time-of-Day (TOD) clock value. If CTOKEN(NO) is specified, the consistency token will be 8 blanks. For a more detailed discussion of consistency tokens, see "Using a Consistency Token" on page 134. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

DATE (EUR)

DATE (ISO)

DATE (JIS)

DATE (LOCAL)

DATE (USA)

If this parameter is specified, the output date format chosen overrides the default format specified at installation time; otherwise, all dates will be returned in the default format specified at installation time. (See the *DB2 Server for VSE & VM SQL Reference* manual for a description of these formats.) This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

NOEXIST

EXIST

If the EXIST parameter is specified, the preprocessor executes normally; that is, it generates modified source code and performs package functions. An error will be generated if objects (such as tables) referenced in statements in the program do not exist or if proper authorization does not exist.

If the NOEXIST parameter is specified, object and authorization existence is not required, and if not found, a warning will be issued. NOEXIST is the default. NOEXIST/EXIST is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

EXPLAIN(NO)

EXPLAIN(YES)

This parameter specifies whether explanatory information for all explainable SQL statements in a package should be produced. EXPLAIN(NO) is the default.

If EXPLAIN(YES) is specified, each explainable SQL statement in the program is explained during preprocessing. If you specify EXPLAIN(YES), an EXPLAIN ALL is executed. The complete set of explanation tables must, therefore, be available. If they are not available, you receive an SQLCODE -649 (SQLSTATE = 42704) and preprocessing is not successful. To interpret the explanation tables, refer to the *DB2 Server for VSE & VM Performance Tuning Handbook* manual. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

NOFOR

This parameter enables you to omit the FOR UPDATE OF clause in the static cursor query statement, and execute positioned updates to any column in the result table for which you have UPDATE authority. It is referred to in this manual as NOFOR support.

Note: This option is also implied if the STDSQL (89) or STDSQL (86) parameter is specified.

NOGRaphic

GRaphic (COBOL and PL/I preprocessors only)

The GRaphic parameter indicates to the preprocessor whether graphic constants can be used in SQL statements and whether DBCS string format should be validated. NOGRaphic is the default.

If GRaphic is specified, the preprocessor accepts SQL statements containing graphic constants, and checks that all strings of DBCS characters are correctly formatted.

If NOGRaphic is specified, the preprocessor does not allow graphic constants in SQL statements, and does not verify the format of strings of DBCS characters.

Note: If the DBCS value in the GLOBAL SQLGLOB parameters is set to YES, the graphic option is not used and preprocessing occurs as though GRaphic had been specified. In addition, the default graphic option becomes GRaphic.

ISOLation (CS)
ISOLation (RR)
ISOLation (UR)
ISOLation (USER)

This parameter lets you specify one of the following isolation levels at which your program runs:

- Specify RR (repeatable read) to have the database manager hold a lock on all data read by the program in the current logical unit of work. This is the default.
- Specify CS (cursor stability) to have the database manager hold a lock only on the row or page of data pointed to by a cursor.
- Specify UR (uncommitted read) to have the database manager allow applications to read data without locking, including uncommitted changes made by other applications.
- Specify USER to have the application program control its isolation level.

See “Selecting the Isolation Level to Lock Data” on page 124 for guidelines on choosing the isolation level for your program. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

Note: If you want to change the ISOLation option, you must recompile (or reassemble) and relink your program after preprocessing it. You must also use CBND to rebuild the package if BIND is specified. Preprocessing alone does not change the ISOLation setting.

KEEP
REVOKE

These parameters are applicable if the program has previously been preprocessed, and the owner has granted the RUN privilege on the resulting package to some other users. Specify the KEEP parameter to have these grants of the RUN privilege remain in effect when the preprocessor produces the new package. Specify the REVOKE parameter to remove all existing grants of the RUN privilege. (These grants will also be removed if the owner of the program is not entitled to grant all the privileges embodied in the program.)

KEEP is the default. KEEP/REVOKE is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

LABEL (*label_text*)

This parameter specifies a label for the package. *Label_text* can be up to 30 characters in length; the default is spaces. This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

LineCount (*integer*)

The parameter determines how many lines per page are to be printed in the output listing. The value *integer* specifies the number of lines per page. The valid range for this value is 10 to 32 767. If no value is specified, or if there is an error in the specification of the LineCount parameter, then the default value of 60 is used.

PACKAGE

NOPACKAGE

If you specify the PACKAGE parameter, the preprocessor performs package functions and creates a package against a local database. PACKAGE is ignored if CHECK is specified; PACKAGE is the default.

If you specify the NOPACKAGE parameter, the preprocessor does not perform package functions and will not create a package. If you specify NOCHECK as well as NOPACKAGE, NOCHECK is ignored. If you specify ERROR as well as NOPACKAGE, ERROR is ignored.

PRint

NOPRint

The PRint parameter specifies that the entire preprocessor modified source listing output is produced. The NOPRint parameter specifies that the preprocessor listing output is suppressed, except for the summary messages that are normally printed at the end. PRint is the default.

PUnch

NOPUnch

The PUnch parameter specifies that the preprocessor modified source output is produced. The NOPUnch parameter specifies that the preprocessor modified source output is suppressed.

REPLACE

NEW

This parameter specifies whether the package being created is new or whether it will replace an existing package that has the same name. If REPLACE is specified and no previous package exists with the same name, no error or warning is issued, and the package is created. REPLACE is the default. If NEW is specified, an error will occur if the package already exists with the same name. REPLACE/NEW is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

SEquence

NOSEquence (C preprocessor only)

If SEquence is specified, the preprocessor searches only columns 1 through 72 of the source file. When NOSEquence is specified, the preprocessor assumes there are no sequence numbers in the input file and it accepts input from columns 1 to 80. SEquence is the default.

Note: In the latter case, you must use the NOSEQ and MARGINS (1,80) C compiler options when compiling the modified source.

NOSQLCA

This parameter allows the user to declare an SQLCODE without declaring all of the SQLCA structure. It is referred to as NOSQLCA support in this manual.

Note: This option is also implied if the STDSQL(89) or STDSQL (86) parameter is specified.

SQLFLAG (SAA)

SQLFLAG (89)

SQLFLAG (89(COMPLETE))

This parameter invokes Flagger, a function that flags those static SQL statements that do not conform to the SQL-89 standard or IBM's

Systems Application Architecture* (SAA*) standard on an SQL dialect. If you specify SAA, it provides syntax checking against the SAA Database Level 1 standard. If you specify 89, it will provide syntax checking against the SQL-89 standard. If you specify 89(COMPLETE), it will provide both syntax and semantics checking against the SQL-89 standard. Note that you cannot check *both* SAA and SQL-89 in the same preprocessor run.

See “Using the Flagger at Preprocessor Time” on page 122 for more details on this facility, including an explanation of the SQL-89 standard.

STDSQL (NO)

STDSQL (89)

STDSQL refers to the SQL Standard that has been implemented in the user’s application program. If NO is specified or the STDSQL parameter is not used, the preprocessor uses DB2 Server for VSE standards. If 89 is specified, functions specific to ANS SQL standard 89 are also provided by the preprocessor. STDSQL(NO) is the default. These functions consist of the following support:

- NOSQLCA
- NOFOR

Note: STDSQL(86) is a synonym for STDSQL(89).

TIME (EUR)

TIME (ISO)

TIME (JIS)

TIME (LOCAL)

TIME (USA)

If this parameter is specified, the output time format chosen overrides the default format specified during installation. If it is not specified, all times will be returned in the default format that was specified during installation. (See the *DB2 Server for VSE & VM SQL Reference* manual for a description of these formats.) This option is stored in the bind file header if BIND is specified and a bind file is successfully created after preprocessing.

Using the Preprocessor Option Member

Instead of specifying all the preprocessing parameters in the preprocessor job you can use an options member. This has several advantages. You can maintain a set of standard options members. They can save time, and ensure consistent use of preprocessing parameters.

You can use a preprocessor options member by including the PrepFile parameter when you are preprocessing. The member can contain only one preprocessor parameter per line. If more are found an error message is returned. Blank lines are ignored, and the parameters may be in either upper- or lowercase. You can insert comments into the options file by placing an asterisk (*) to the left of the comment. Everything to the right of the asterisk is ignored. The file is a VSE source member.

```

* prep parameters for program SAMPLE
  ISOL(CS)           *cursor stability isolation level
  TIME(ISO)
  BLOCK             *indicate inserts and retrieves in groups

```

Figure 41. The preprocessor option file example.

Using the Flagger at Preprocessor Time

The Flagger is invoked at preprocessor time by the optional parameter SQLFLAG. It provides an auditing function on the static SQL statements in the host program. This function is independent of the other preprocessor functions, and has no bearing on whether the preprocessor run will complete satisfactorily.

The audit compares the static SQL statements with the SAA standard or the SQL-89 standard. SQL-89 is a collective term that implies support of SQL as defined by the Federal Information Processing Standards (FIPS) 127-1. It includes:

- ANSI X.3.135-1989 (without the Integrity Enhancement feature)
- ANSI X.3.168-1989
- ISO 9075-1989 (without the Integrity Enhancement feature)

In addition to basic syntax checking against SQL-89, Flagger optionally performs semantics checking against SQL-89. This includes some integrity checking between the SQL statements and the database. For example, it checks:

- Whether a statement contains column names or table names that do not currently exist.
- Whether a statement contains ambiguity among column names, such as an unqualified name for a column that exists in more than one of the tables in the query.
- Whether a statement contains inconsistencies between the data types of the host variables and their corresponding table columns.

Any statements that do not conform to the standards are flagged in the form of information messages in the preprocessor output listing. Flagger, however, does not force you to comply with the standards. The purpose of Flagger is to provide guidance for those users who want to conform to these standards, so that they can have SQL consistency across operating environments.

Note: The DB2 Server for VSE product is a superset of the SQL-89 standard without the Integrity Enhancement feature. For example, the datetime data types are not part of SQL-89 and the CONNECT statement is not part of SQL-89. The use of extensions such as these will generate information messages for deviations from the standard specified in the SQLFLAG parameter.

The Flagger messages generated at preprocessor time range from ARI5500 to ARI5599, and are further classified as follows:

1. ARI5500-ARI5539 and ARI5570-ARI5599 are information messages that indicate that an extension to the SQL-89 standard (nonconformance) has been found. These start with "FLAGGER message."

2. ARI5540-ARI5569 are warning messages that indicate a failure on the part of Flagger itself.

In this event, SQL-89 semantics checking will be turned off and its syntax checking may or may not be turned off, depending on the nature of the failure. However, the preprocessor run itself will continue, and any inconsistencies discovered by Flagger prior to the failure will be included in the output listing of the run.

Using the CICS/VSE Translator

CICS/VSE provides translators for C, COBOL, PL/I, and assembler language programs to convert CICS/VSE statements to CICS/VSE calls, similar to the function performed by the DB2 Server for VSE preprocessors. When program modules include both SQL and CICS/VSE statements, the appropriate CICS/VSE translator and the appropriate DB2 Server for VSE preprocessor must be run before compiling the language. This replaces the CICS/VSE and SQL statements with appropriate host language statements that invoke the CICS/VSE or database manager.

SQL statements may contain certain encoding, in quoted strings, that would not be properly bypassed by the CICS/VSE translators when scanning for CICS/VSE statements. For example,

```
"...EXEC CICS ..."  "...'..."
```

When using the QUOTE option in COBOL, the same problem may occur for single-quoted strings. Also, when SQL statements contain DBCS constants, a similar problem may arise because DBCS constants may contain single quotation marks as part of the double-byte character set.

If a CICS/VSE translator is run before the DB2 Server for VSE preprocessor, these problems may occur in the form of unmatched quotation marks from a CICS/VSE standpoint, because a translator does not allow for SQL statements embedded in programs. Therefore, you should run the DB2 Server for VSE preprocessor before running a CICS/VSE translator. This will ensure that the SQL statements are commented out before a CICS/VSE translator processes the program.

Currently, there is a problem with running the DB2 Server for VSE preprocessor before a CICS translator. The preprocessor's output goes to SYSPCH as 81-byte records; a CICS/VSE translator, however, accepts only 80-byte input. For COBOL and PL/I programs this is not a severe problem, because these preprocessors append the stacker select and punch control character as the eighty-first byte. While SYSPCH is unblocked, CICS ignores the extra byte.

For assembler and C, however, the card punch control character is appended as the first byte. In this case it is necessary to process the DB2 Server for VSE assembler preprocessor output with a utility (OBJMAINT) to eliminate the leading byte before processing by a CICS translator. See Figure 46 on page 138.

These problems can also be avoided by placing the SQL statements in a separate module from the one containing CICS statements.

Improving Performance Using Preprocessing Parameters

When preprocessing your program, you can specify two performance parameters, the BLock/ NOBLock option and the ISOLation level option. discussed under “Preprocessing the Program” on page 108. These options are specified under the PARM keyword of the job control EXEC statement. The format and use of these parameters was discussed earlier in this chapter. The next section discusses when you would want to specify each of these options.

(Other performance considerations are discussed in the *DB2 Server for VSE Database Administration* manual.)

Selecting the Isolation Level to Lock Data

The database manager puts locks on data that your program works with, to keep other users from reading or changing that data. You can specify how long the system holds the lock on data. You can specify either to lock **all** the data that the current logical unit of work (LUW) has read, to lock just the row or page of data that a cursor is currently pointing to, or to not lock any data being read. This is called specifying the *isolation level* of the lock.

The isolation level used by an application is set using the ISOLation preprocessing parameter. On SELECT, SELECT INTO, INSERT, searched UPDATE, and DELETE statements, the WITH clause may be specified to override the value specified on the preprocessing parameter.

If you choose to put a lock on all the data that your program's current LUW has read, this is called specifying *isolation level repeatable read*. Repeatable read locks are held until the end of the LUW. If you choose to put a lock on just the row or page of data that your cursor is pointing to, then you are specifying *isolation level cursor stability*. With cursor stability locking, when the cursor moves, the system frees all the data previously read by the program that was held by the lock. If you choose not to lock the data that your program will read, this is called specifying *isolation level uncommitted read*. With uncommitted read, no locks are held on the data being read, and as a result, the data can be changed by other applications.

Both repeatable read and cursor stability provide you with the following data isolation from other concurrent users:

- Your LUW cannot modify or read any data that another active LUW has modified. Similarly, if your LUW has modified some data, no one else can modify or read that data until your LUW has ended. Modify means to apply INSERT, DELETE, UPDATE, or PUT commands; READ means to apply SELECT or FETCH commands.
- If your LUW has a cursor pointing to a row of data, no other LUW can modify that data. Similarly, your LUW cannot modify a row to which another user has a cursor pointing.

In addition to the above, repeatable read locking provides you with the following data isolation from other concurrent users:

- No other LUW can modify any row that your active LUW has read. Also, you cannot modify any data that another active LUW, specifying repeatable read, has read.
- You do not have to worry about your data being changed between reads, as long as you do not end your LUW between those reads.

This extra isolation has its drawbacks, however. When you specify repeatable read for data in public dbspaces with PAGE or ROW level locking, you reduce the concurrency of the data. This means that other users may be locked out from the data for a long time, causing delays in their programs' executions.

If you specify cursor stability instead, you reduce these locking problems by making the data more available. With this isolation level, the system does not hold the locks as long. After a cursor has moved past a row or page of data, the lock on that data is dropped. This increases concurrency so that other users can access data faster.

Cursor stability can, however, cause some data inconsistencies. For instance:

1. If a user's LUW reads data twice, it can get different results. This could happen if another user modifies the data and commits the changes between read operations.
2. A modification based on a prior reading can be incorrect. This can occur if another LUW modifies the rows that a user has read and commits the changes before that user can do the modification. (Note that when the user is retrieving data in application programs, the only row that is safe from modification is the one that is currently being pointed to by a cursor.)
3. If an SQL statement in the user's LUW is traversing a table by way of an index, the user might find the same row twice. (This case applies to FETCH cursors, searched INSERT by way of subselect, and searched UPDATE with subselect that traverse a table by way of an index.) This can occur because, after the user's statement reads the row the first time, another user can update the column value that is indexed and commit the change. The change could cause the committed row to be *ahead* of the row currently being retrieved by the statement. The first user's statement would then find the row again with its updated index column value.
4. If an SQL statement in the LUW is traversing a table by way of an index, it can fail to find a row (or rows) even if the row meets the selection criteria. (This situation applies to FETCH cursors, Searched DELETE, Searched INSERT by way of the subselect, and Searched UPDATE by way of the subselect that traverse a table by way of an index.) This can occur because while the LUW is reading, another user modifies the indexed column in the row and commits the change (as above). The change could cause the committed row to be *behind* the row the user's statement is currently reading. Thus, the statement would not find the row, even if the row met the selection criteria.
5. If you enter a SELECT statement to retrieve a single row, a cursor is opened when the system processes the statement and is closed when the row is returned. All PAGE and ROW level locks are released when the cursor is closed; therefore, no locks are held after the row is returned. For single-row processing using a SELECT statement with a fully qualified unique index, a cursor is not opened and again no locks are held once the row has been returned. As a result, applications which update a selected column based on the values retrieved may have unexpected results because the lock was not held for the duration of the LUW. For example:

```

HOST_EMPNO = '000250'
EXEC SQL SELECT SALARY                               /* HOST_SALARY is 19180      */
      INTO :HOST_SALARY
      FROM EMPLOYEE
      WHERE EMPNO = :HOST_EMPNO;
HOST_SALARY = HOST_SALARY + 1000; /* HOST_SALARY increased to 20180 */

EXEC SQL UPDATE EMPLOYEE                             /* UPDATE SALARY in EMPLOYEE */
      SET SALARY = :HOST_SALARY; /* TABLE with HOST_SALARY */
      WHERE EMPNO = :HOST_EMPNO;
EXEC SQL SELECT SALARY                               /* HOST_SALARY may not be 20180 */
      INTO :HOST_SALARY                             /* because lock was not held for*/
      FROM EMPLOYEE                                 /* the duration of the LUW      */
      WHERE EMPNO = :HOST_EMPNO;
COMMIT WORK;

```

In the previous example, it is possible that two or more users could read the salary column with the same value at approximately the same time. They would then each increment the number and issue the UPDATE statement. The second user would wait for the first user's update to finish, and then overwrite it with the same number.

Unlike RR or CS, uncommitted read does not provide any data isolation from other concurrent users. Like CS though, concurrency is improved, although at the risk of data inconsistency. UR can cause similar data inconsistencies as those described for CS and should only be used when it is not necessary that the data you are reading be committed.

An application using isolation level UR is still restricted to access only data for which it has authorization. However, because it will be able to read uncommitted changes, it will be able to read additional rows which an application, with the same authorization but using RR or CS, could not. This is illustrated by the following example.

Rows of table:	Scenario: 1
A	U1 reads (using UR) A B
B	U2 inserts D
C	U1 continues reading C D E
<---D	U2 rolls back
E	-- U1 has read a non-existent row
	Scenario: 2
	U1 reads (using CS) A B
	U2 inserts D
	U1 continues reading C, must wait to read D
	U2 rolls back
	U1 continues reading E
	Note: In scenario 1, U1 has read an extra row which U1 in scenario 2 could not.

When should each of these options be chosen for your program? Usually, you should specify repeatable read locking. Only use cursor stability if your program causes or will cause locking problems. For instance, you would probably want to use cursor stability for transactions that perform terminal reads without performing a COMMIT or ROLLBACK, or programs that do bulk reading, because it is handy for programs that browse through large amounts of data. For programs that perform commits or rollbacks before issuing terminal reads, you should use repeatable read

locking, because they probably will not cause locking problems. Also, any application that needs to protect itself against updates should also use repeatable read locking. For programs where concurrency is wanted, for example, data being queried simultaneous to being updated, you would use uncommitted read locking. Of course, this would be for applications where data integrity was not important because the data being read may not necessarily have been committed.

You can also mix isolation levels, to have your program set, change, and control its own isolation level as it is running. You can specify mixed isolation level with the USER option of the ISOLation preprocessor parameter, as detailed under "Preprocessing the Program" on page 108.

If you choose this option, your program must pass the isolation level value to the application server by a program variable. It *must* declare a one-character program variable and *must* set this variable to the desired isolation level value before executing SQL statements. For repeatable read, your program should set this variable to R; for cursor stability, the variable should be set to C; and for uncommitted read, the variable should be set to U. The program can change the variable at any time so that subsequent SQL statements are executed at the new isolation level value. However, if your program changes the isolation level while a cursor is OPEN, the change does not take effect for operations on that cursor until it has been closed and opened again. That is, until the cursor is closed all operations on that cursor are executed at the isolation level value that was in effect when the cursor was opened. Note that the changed isolation level *will* be used (without error) for SQL statements not referencing the opened cursor.

If the program sets the isolation level variable to a value other than C, R or U, or if it fails to initialize the variable, the system stops execution and returns an error code in the SQLCA.

Figure 42 shows the isolation level variable name for each of the host languages.

<i>Figure 42. Variable Names for Specifying Mixed Isolation Levels</i>		
Host Language	Variable Name	Example
assembler	SQLISL	SQLISL DS CL1
C	SQLISL	char SQLISL;
COBOL	SQL-ISL	01 SQL-ISL PIC X(1).
FORTRAN	SQLISL	CHARACTER SQLISL
PL/I	SQLISL	DCL SQLISL CHAR(1);

Note: If you forget to declare the isolation level variable in a PL/I program, the PL/I compiler issues an informational message which can, in some environments, be suppressed.

Isolation level cursor stability or uncommitted read only has meaning for data in public dbspaces with ROW or PAGE level locking. Data in private dbspaces or in public dbspaces with DBSPACE level locking always uses repeatable read isolation. However, programs which access such data and do not require repeatable read should be preprocessed with cursor stability or uncommitted read. The data concurrency requirements might change and cause the data to be moved to a public dbspace with PAGE or ROW level locking. In this case, the program

would not need to be reprocessed to run at isolation level cursor stability or uncommitted read .

To use the features of CS or UR, data must reside in public dbspaces with PAGE or ROW level locking. DML statements against private dbspaces or public dbspaces with PAGE or ROW level locking under isolation level CS or UR are handled the same as if isolation level RR were used.

When the system uses a dspace scan (that is, does not use an index) to access a table in a dspace with ROW level locking using isolation level cursor stability, the effect is the same as repeatable read. That is, no other LUW can update the table until the logical unit of work performing the dspace scan ends. Also, if an LUW is updating a table, another LUW (using cursor stability) cannot access that table with a dspace scan until the updating LUW ends. This reduced concurrency for dspace scans does not apply to tables in dbspaces with PAGE level locking, or to accessing through indexes. Because most database accesses will typically use indexes, the reduced concurrency caused by dspace scans should not occur frequently.

The isolation level specification affects UPDATE and DELETE processing as well as SELECT processing. For UPDATE and DELETE processing, the system acquires UPDATE locks. UPDATE locks can be acquired for both cursor stability and repeatable read isolation level settings. If the user actually wants to update or delete the data, the UPDATE lock is changed to an EXCLUSIVE lock; otherwise, the UPDATE lock is changed to a SHARE lock.

Note the following about UPDATE LOCKS:

- They are used for page or row locking, but not for dspace locking.
- They apply to index pages or index keys only for the Searched DELETE statement.
- For Positioned DELETE processing, the named cursor must have been declared in the FOR UPDATE clause of the DECLARE CURSOR statement.
- For FETCH processing that uses repeatable read isolation level, these locks are acquired only if certain predicates are present in the statement. See the *DB2 Server for VSE Database Administration* manual for more information.

Internally generated SELECT, UPDATE, or DELETE statements use cursor stability locking no matter what the isolation level is set to. (See “Enforcing Referential Integrity” on page 242 for information on these statements). Conversely, data definition statements such as CREATE, ACQUIRE, or GRANT, use repeatable read locking no matter what the isolation level is set to. These statements, therefore, should not play a role in your choice of isolation level.

Note: Catalog access for SQL statement preprocessing is also always done with repeatable read locking.

Using the Blocking Option to Process Rows in Groups

You can insert and retrieve rows in groups or blocks, instead of one at a time. This is called specifying the *blocking* option. When you specify the blocking option, improves performance for DB2 Server for VSE application programs that:

- Execute in multiple user mode, and
- Retrieve or insert multiple rows.

You can specify the blocking option as a DB2 Server for VSE preprocessor parameter, or as an option on the CREATE PACKAGE statement. After a program has been preprocessed with the blocking option, all *eligible* cursor SELECTs and all *eligible* cursor INSERTs within the program are blocked. You do not have to specify a block size or block factor. The block size for inserts and SELECTs is automatically fixed.

The programs that would benefit the most from blocking are those that do multiple-row inserts (with PUT statements) or multiple-row SELECTs (with FETCH statements). In both cases, a cursor must be defined. (See “Retrieving or Inserting Multiple Rows” on page 32 for more information on cursors.) Thus, a general rule for blocking is USE BLOCKING FOR PROGRAMS THAT DECLARE CURSORS.

A program can use either PUT or FETCH statements without being sensitive to whether the system is blocking. These statements work regardless of whether you specified the blocking option. What information is returned in the SQLCA after each PUT or FETCH, however, depends on whether blocking is in effect or not.

Remember that when you preprocess a program with the blocking option, *all* eligible INSERT and SELECT cursors are blocked. You cannot specify blocking for just INSERTs or for just SELECTs. If you specify the blocking option, it automatically applies to both.

When are INSERT or SELECT statements not eligible for blocking? The system sometimes overrides blocking for a particular cursor because of storage limitations in the database partition, or because of SQL statement ineligibility.

- DECLARE CURSOR...FOR UPDATE
- Any DECLARE CURSOR statement with a related *select-statement* containing a long string
- Any DECLARE CURSOR statement that has a subsequent DELETE...WHERE CURRENT OF statement
- Any DECLARE CURSOR statement that has a corresponding UPDATE...WHERE CURRENT OF statement and the program is preprocessed with NOFOR support.

The system also disqualifies blocking if it cannot fit at least two rows into a block. (The number of rows that fit into a block may differ from one PUT/FETCH statement to the next, even when such statements operate on the same table.)

The system does *not* halt the program when it overrides blocking. Instead, in each of the above cases, it sets a warning flag in the SQLCA. The warning can be detected by using WHENEVER SQLWARNING in the program. See “Using the Automatic Error-Handling Facilities” on page 147 for more information on the SQLCA and the SQL WHENEVER declarative statement.

Note: The DECLARE CURSOR... statement can also be written without the FOR UPDATE OF clause, even though positioned updating is subsequently done. (This is allowed when NOFOR support is invoked at preprocessor time.) In this case, blocking is also ineligible.

The system also overrides blocking for all programs running in single user mode. In this instance, the system does *not* usually return a warning to the SQLCA. A warning is returned to the SQLCA for programs running in single user mode if:

- The program is preprocessed with the BLock option
- An SQL statement that is being processed dynamically (with PREPARE) is disqualified for blocking.

The DBS Utility may get blocking ineligible warnings when it is run in single user mode because it is preprocessed with the BLock option, but uses PREPARE to process SELECT statements.

Note: Always CLOSE a cursor before issuing a COMMIT statement, especially when blocking. If you commit changes before closing an insert cursor that is being blocked, you receive an error.

Imposing Blocking Restrictions

1. The length of host variables in the SQLDA or *host_variable_list* cannot be changed after the first FETCH or PUT when blocking.
2. The data type of host variables in the SQLDA or *host_variable_list* cannot be changed after the first FETCH or PUT when blocking.
3. The number of data elements in the *host_variable_list* or SQLDA cannot be changed after the first FETCH or PUT when blocking.
4. If a COMMIT is issued while a blocking PUT cursor is open, an error occurs.

When blocking is active, a single SQLCA is returned with each block of rows. This SQLCA is returned to the application program with the last row in the block. However, for the final block of rows, the FETCH that returns the “not found” condition (SQLCODE = +100 and SQLSTATE='02000') will return the SQLCA. (For more information on SQLCA refer to “Using the SQLCA” on page 148). This has the following implications for application programming:

- No warning conditions are returned to the application until the SQLCA is returned.

For example, if SQLWARN3 is set (to indicate that the application has fewer target variables in the INTO clause than the number of items in the SELECT list), the application will not be notified until either the last row in a block or the “not found” condition is returned.

- If SQLWARN1 (truncation occurrence) is set, it is impossible to tell from the SQLCA information which row (or rows) in a block caused the warning condition. However, if the application resets the indicator variable to 0 before each fetch, and then examines the indicator variable after each fetch, truncation can be detected on an ongoing basis.

Using the Blocking Option in DRDA Protocol: If the DB2 Server for VSE database manager is acting as an application server to a non- DB2 Server for VSE application requester during a FETCH operation, the application server stores reply messages in blocking buffers. The size of the buffer is less than or equal to the block size, except when the data rows are longer than the block size.

Using the INCLUDE Statement

Including External Source Members

The inclusion of external source members is indicated to the DB2 Server for VSE preprocessor by an embedded SQL statement, the INCLUDE statement, in the user's source code. This statement can appear anywhere that an SQL statement can appear, and indicates within the source code where the external source is to be placed. The syntax for the INCLUDE statement is as follows:

```
▶▶—INCLUDE—text_file_name—◀◀
```

where *text_file_name* identifies the external source member. The *text_file_name* is a 1 to 8 character identifier and cannot be delimited by double quotation marks. The first character must be a letter (A-Z), \$, #, or @; the remaining characters must be letters, digits (0-9), \$, #, @, or underscore (_) unless further restricted by the operating system. Also, *text_file_name* cannot be SQLCA or SQLDA, as these are special INCLUDE keywords.

The statements contained in the external source specified by may be host language statements or SQL statements (except for another INCLUDE statement). INCLUDE statements may not be nested, but the external source may contain INCLUDE SQLDA or INCLUDE SQLCA statements.

Note: The INCLUDE statement can be in an SQL DECLARE section, or the entire SQL DECLARE section can be within external source members.

Including a Library Source

The INCLUDE statement may be used to obtain secondary input from a VSE source member.

INCLUDE causes input to be read from the specified source member until the end of that source member is read. At this time, SYSIPT input resumes. (File records representing the source statements must be unblocked, fixed-length, 80-character records.) The source member must be cataloged as the following source types:

- A - assembler
- B - C
- C - COBOL
- G - FORTRAN
- P - PL/I

If the INCLUDE statement specifies a source member that is not cataloged for the appropriate source member type, an error results.

Source member input must not contain preprocessor INCLUDE statements other than INCLUDE SQLDA or INCLUDE SQLCA, although it may contain both host language and DB2 Server for VSE statements. If an INCLUDE statement is encountered, an error will result.

Within the INCLUDE statement, *text_file_name* specifies a "source member" in the DB2 Server for VSE library. The source member type is determined by the preprocessor that is invoked.

The *text_file_name* specification must not contain the source member type identifier; the source member type will be based on the aforementioned preprocessor.

Compiling the Program

After you successfully preprocess your program, you can compile it using your normal host language compiler, and create an object deck. Use the modified source output of the DB2 Server for VSE preprocessor as input to the compiler. By preprocessing, you have already done all the translating that the program needs for the database manager. However, if the program is an on-line program, the CICS statements have to be “translated” before the program is compiled. Just use the new code that you got after you preprocessed. Compile this code as you would any other program, using the usual compilers.

This book does not cover the specifics of compiling your host-language code. However, there are several special rules for SQL programs, depending on the host language, that you must follow:

- If your PL/I application program contains DBCS data, you must specify the GRAPHIC option for the compiler. If your COBOL application program contains DBCS data or is reentrant, the output of the DB2 Server for VSE preprocessor must be processed by the COBOL II Release 2 (or later) program.
- If the QUOTE option is used for the DB2 Server for VSE COBOL preprocessor, it should also be used for the COBOL compiler.
- If the NOSEQUence option is used for the DB2 Server for VSE C preprocessor, the NOSEQ and MARGINS (1,80) options must be used with the C compiler.

Link-Editing and Loading the Program

After compilation, programs must be link-edited and loaded before they can be run.

Link-Editing the Program with Supplementary Information

Including Relocatable Modules

To allow your program to communicate with the database manager, you must link-edit your program with one or more DB2 Server for VSE relocatable modules. One of these relocatable modules is called the *resource adapter stub*. Every DB2 Server for VSE application program must be link-edited with this stub. FORTRAN and COBOL programs need to be link-edited with additional relocatable modules. Also, depending on the nature of your program, you may have to link-edit with others. For instance, when link-editing a module that uses CICS/VSE, you may have to INCLUDE a CICS/VSE module immediately following the PHASE system control statement and before the EXEC ASSEMBLY system control statement. Whether you INCLUDE this module depends on your host language.

When you load your program, the VSE linkage editor automatically links your program to all modules that you included. The linkage editor also resolves virtual storage addresses between files.

Including CICS/VSE Procedures

When link-editing a module that uses CICS/VSE, you should include the following procedures for your application:

- For all assembler applications, you must have an INCLUDE for CICS/VSE module DFHEAI immediately following the PHASE system control statement and before the EXEC ASSEMBLY system control statement. DFHEAI must be the phase entry point.
- For all C applications, you must have an INCLUDE for CICS/VSE module DFHELII. This module must be the phase entry point.
- For all COBOL applications, you must have an INCLUDE for CICS/VSE module DFHECI immediately following the PHASE system control statement and before the EXEC IGYCRCTL (EXEC FCOBOL in OS/VS COBOL) system control statement. DFHECI must be the phase entry point.
- For all PL/I applications, you must have an INCLUDE for CICS/VSE module DFHPL11 immediately following the PHASE system control statement and before the EXEC PLIOPT (EXEC IEL1AA in IBM PL/I for VSE) system control statement. DFHPL11 must be the phase entry point.

See Figure 46 on page 138 for an example of these procedures.

Including Extra Linkage Modules

When link-editing *any* DB2 Server for VSE application, you must include some extra linkage modules *following* the application program module that uses the database manager. Use these extra modules as indicated below:

- For all batch and ICCF applications, you must include the linkage module ARIPRDID.
- For all online (CICS/VSE) applications, you must include the linkage module ARIRRTED.

In addition to the modules listed above, some of these may be required:

- For all programs written in COBOL (regardless of whether they are batch, ICCF, or online), you must also include the module ARIPADR4. Include ARIPADR if your program was preprocessed prior to SQL/DS Version 2 Release 2. (You need not include it if VSE autolink is used.)
Note: If you use a COBOL SORT in your program, include ARIPADR4 (or ARIPADR) and ARIPRDID (or ARIRRTED) before the compile step.
- For all programs written in FORTRAN, you must also include the modules ARIPEIFA and ARIPSTR. If the FORTRAN program uses the module ARISSMF, this file must also be included. See the discussion of the SQLCA in Chapter 5, “Testing and Debugging” on page 145.
- For all programs that will include the DBS utility (regardless of whether they are batch, ICCF, or online), you must also include the modules ARISYSDD, ARIDSQLA, and ARIDDFP.
- All programs (except FORTRAN programs) that use the module ARISSMA, must also link-edit this module. See the discussion of the SQLCA in Chapter 5, “Testing and Debugging” on page 145 for further information.

If you receive an unresolved external reference message for a module name that begins with ARI or SQL, check the link process to ensure that all required extra linkage modules are included.

Some of these modules contain entry points with names that are different from the module name. The code generated by the DB2 Server for VSE preprocessor can reference one of these entry points, depending on the SQL statements in your application.

Running the Program

Using a Consistency Token

Consistency tokens ensure that a program's load module and the database package are used together. When preprocessing, you can instruct the preprocessor to place a consistency token in both the load module and the package (see CTOKEN parameter 117). If the two tokens do not match, the application server prevents the program from running.

Note: If you inadvertently forget to compile or link-edit a new version of a program, you can run an old version of a program with a new version of the package. Conversely, with multiple application servers, you can inadvertently run a new version of a program with an old version of the package. In either situation, you will probably get program errors or incorrect results if you have not used consistency tokens.

Loading the Package and Rebinding

The package that the preprocessors or CBND stored carries out the SQL request. When the application server loads the package, it checks that the package is still valid. A package may not be valid if one of its dependencies has been dropped. For example, some index that the package uses may have been dropped.

Packages are also invalidated when primary keys and referential constraints are added to, dropped from, activated, or deactivated on tables that the modules depend on. The following rules apply:

- If a primary key is added, dropped, activated, or deactivated, all packages that have a dependency on the parent table will be invalidated. This includes any tables that have a foreign key relationship with the parent table.
- If a foreign key is added, dropped, activated, or deactivated, all packages that have a dependency on the dependent table or parent table will be invalidated.

The system has an internal change management facility that keeps track of whether packages are valid or not. If a package is valid, the system begins running the program; if the package is not valid, the system tries to re-create it. The original SQL statements are stored with the package when you preprocess the program. The system uses them to automatically bind the program again. It does this dynamically (that is, while it is running). If the *rebinding* works, a new package is created and stored in the database and the system then continues execution of the program. If the rebinding does not work, an error code is returned to the program in the SQLCA, and the program stops running.

A successful rebinding has no negative effect on your program except for a slight delay in processing your first SQL statement. To minimize this delay, you can use the DBSU REBIND PACKAGE command to rebind the invalid package after it has been invalidated, but before it is executed. See the *DB2 Server for VSE & VM Database Services Utility* manual for information on this command.

Running by Mode

Using Multiple User Mode

The job control for multiple user mode executes the application program; the application server must already be initialized. The application program communicates with the application server, which resides in a separate partition, through the DB2 Server for VSE linkage module (ARIPRDID or ARIRRTED). The linkage module is given control for each SQL statement in the application program.

You can invoke the application with standard job control statements or CICS/VSE procedures. It is recommended that you specify SIZE=AUTO. (Specify SIZE=750K for PL/I programs.) Figure 43 shows job control for invoking programs in multiple user mode:

```
// JOB USER PROGRAM
// EXEC PGM=MYPROG,SIZE=AUTO
/*
/ &
```

Figure 43. Invoking a User Program in Multiple User Mode

Using Single User Mode

The job control for single user mode invokes the application server and passes to the system the name of the application to be run. The system then loads and invokes the application program. Once the single user mode application receives control, it accesses the application server in the same way as multiple user mode applications. The system supports the VSE register save conventions. See the *DB2 Server for VSE System Administration* manual for more about single user mode execution using the VSE register save conventions.

The *DB2 Server for VSE & VM Operation* manual lists all the initialization parameters you can specify when you start the system in single user mode. A system programmer can also determine the best initialization parameters for your installation and pass them on to you.

Figure 44 shows job control for invoking programs in single user mode. The SIZE=AUTO specification is required. (Specify SIZE-AUTO, 750K for PL/I programs.)

```
// JOB SINGLE
// EXEC PGM=ARISQLDS,SIZE=AUTO,PARM='SYSMODE=S,PROGNAME=name'
/*
/ &
```

Figure 44. Invoking a User Program in Multiple User Mode

If your program or the system abnormally ends, you may receive a mini-dump (depending on what initialization parameters were specified when the system was started). Mini-dumps are described in the *DB2 Server for VSE System Administration* manual.

Running under CICS/VSE Support

DB2 Server for VSE-CICS/VSE support must be initialized before applications can be run as CICS/VSE transactions. Usually the CICS/VSE support is initialized when the system is started. Once the support is started, you can use normal CICS/VSE procedures to invoke transactions that use the system. See the *DB2 Server for VSE System Administration* manual for more information on starting and stopping CICS/VSE support.

Accessing Other DB2 Family Application Servers

If your VSE/ESA has been initialized as a VSE/ESA guest system under VM, your (multiple user mode) VSE application can access a DB2 Server for VM application server. Therefore, if the application was designed to run in single user mode it must be converted to multiple user mode. This conversion can be accomplished by modifying the job control language (JCL). See the *DB2 Server for VSE System Administration* manual for more information on VSE Guest Sharing.

Your online CICS/VSE application can also use the DRDA protocol to access other DB2 Family application servers, including DB2 Server for VM, DB2 Universal Database Server for OS/390, or DB2 Universal Database (UDB). Before doing this, the DRDA requester code must be installed on your VSE system. Also each DB2 Family server to be accessed must be defined to the DB2 Server for VSE online resource adapter as a remote application server. See the *DB2 Server for VSE System Administration* manual for more details on defining remote application servers.

Installing Applications that Access the Database Manager

Installing a Batch Application

Installing a batch application involves running the application through a DB2 Server for VSE preprocessor. The preprocessor output must be compiled (or assembled), and the object decks must be link-edited. Refer to Figure 45 on page 137 for generic JCL for these tasks.

```

// MYPROG program name
*
// DLBL IJSYSPH,'PREPROCESSOR.OUTPUT',0 *-- PREPROCESSOR output
// EXTENT SYSPCH,..... *--
  ASSGN SYSPCH,.... *-- SYSPCH assignment
*
// LIBDEF ..... *-- Library definitions
*
// ASSGN SYSxxx,... *-- Preprocessor workfiles
*
// DLBL SQLGLOB,....,DISP=(OLD,KEEP) *-- SQLGLOB parm file
// DLBL SQLBIND,....,DISP=(OLD,KEEP) *-- BIND output file
// DLBL BINDWKF,.... *-- BIND work file
*
// EXEC ARIPRPx,SIZE=AUTO,PARM='USERID=SQLDBA/SQLDBAPW,PREPNAME=MYPROG,*
      KEEP,..... ' *-- Invoke DB2 for VSE PREPROCESSOR
*
* MYPROG input here if SYSRDR and SYSIPT assigned to same device
*
/*
  CLOSE SYSPCH,00D *-- Close SYSPCH
*
// DLBL IJSYSIN,'PREPROCESSOR.OUTPUT',0 *-- Input File
// EXTENT SYSIPT,..... *--
  ASSGN SYSIPT,.... *--
*
// OPTION CATAL ..... *--
  ACTION MAP *--
  PHASE MYPROG,* *--
// EXEC compiler *-- Compile
  INCLUDE ARIPRPID *-- DB2 for VSE Resource
                        Adapter stub
  INCLUDE ..... *-- Include runtime routines
*
// EXEC LNKEDT *-- Link Edit
  CLOSE SYSIPT,00C *-- Reset SYSIPT
/*
/&

```

Notes:

1. JCL must be changed to specify the correct DASD extents, Device address, Compiler references & Library definitions
2. Replace ARIPRPx with the preprocessor name
3. See Figure 36 on page 110 for a list of preprocessor work files
4. See "Link-Editing and Loading the Program" on page 132 for a complete list of modules to be included.

Figure 45. Creating an Object Deck and Phase for a Batch Program (Multiple User Mode)

Installing an Online CICS/VSE Application

Two steps are required to install an online application:

1. You must run the application through a DB2 Server for VSE preprocessor. If the application uses the EXEC CICS interface, it must also be run through the CICS/VSE translator. The CICS/VSE translator output must be compiled (or assembled), and the object decks must be link-edited.
2. CICS/VSE must be made aware of the application. You must define the CICS/VSE transaction identifier to be used to activate the application, and define the phase associated with the transaction identifier. Refer to Figure 46 on page 138 for generic JCL for these tasks.

As indicated in Step 2, every on-line application must also have normal PCT and PPT entries generated for CICS/VSE or have these entries defined using the

Resource Definition Online (RDO) facility. For more information about creating these entries, see a system programmer or refer to the *CICS/VSE Resource Definition (Online)* manual. For resource synchronization, specify DTB=YES for each DFHPCT statement that defines a transaction that will access the database manager.

```

// MYPROG program name
// DLBL IJSYSPH,'PREPROCESSOR.OUTPUT',0 *-- PREPROCESSOR output
// EXTENT SYSPCH,..... *--
  ASSGN SYSPCH,... *-- SYSPCH assignment
// LIBDEF ..... *-- Library definitions
// ASSGN SYSxxx,... *-- Preprocessor workfiles
// DLBL SQLGLOB,.....,DISP=(OLD,KEEP) *-- SQLGLOB parm file
// DLBL SQLBIND,.....,DISP=(OLD,KEEP) *-- BIND output file
// DLBL BINDWKF,..... *-- BIND work file
// EXEC ARIPRPX,SIZE=AUTO,PARAM='USERID=SQLDBA/SQLDBAPW,PREPNAME=MYPROG,*
      KEEP,..... ' *-- Invoke DB2 for VSE PREPROCESSOR
*
* MYPROG input here if SYSRDR and SYSIPT assigned to same device
*
/*
  CLOSE SYSPCH,00D *-- Close SYSPCH
*
* For ASSEMBLER only, convert to 80 byte input format for
* CICS Translator, dropping the stacker select prefix added
* to SYSPCH above. See notes for details
*
// DLBL IJSYSIN,'PREPROCESSOR.OUTPUT',0 *--} For ASSEMBLER replace
// EXTENT SYSIPT,..... *--} with statements in the
  ASSGN SYSIPT,... *--} notes section
*
// DLBL IJSYSPH,'TRANSLATOR.OUTPUT',0 *-- CICS Translator output
// EXTENT SYSPCH,..... *--
  ASSGN SYSPCH,... *--
*
// EXEC DFHEXP1$,SIZE=400K *-- CICS Translator
  CLOSE SYSIPT,00C *--
  CLOSE SYSPCH,00D *--
*
// DLBL IJSYSIN,'TRANSLATOR.OUTPUT',0 *-- CICS Translator output
// EXTENT SYSIPT,..... *-- (input to the compiler)
  ASSGN SYSIPT,... *--
*
// OPTION CATAL ..... *--
  ACTION MAP *--
  PHASE MYPROG,* *--
  INCLUDE DFHxxxx *-- CICS host language module
// EXEC compiler *-- Compile
  INCLUDE ARIRRTED *-- DB2 for VSE Resource
      Adapter stub
  INCLUDE ..... *-- Include runtime routines
// EXEC LNKEDT *-- Link Edit
  CLOSE SYSIPT,00C *-- Reset SYSIPT
/*
/&

```

Figure 46 (Part 1 of 2). Creating an Object Deck and Phase for an Online Program (Multiple User Mode)

Notes:

1. JCL must be changed to specify the correct DASD extents, Device address, Compiler/Translator references and Library definitions
2. Replace ARIPRPx with the preprocessor name
3. Replace DFHEXP1\$ with the CICS Host Language Translator
4. See Figure 36 on page 110 for a list of preprocessor work files
5. See “Link-Editing and Loading the Program” on page 132 for a complete list of modules to be included
6. For assembler and C, the following statements are to replace the three statements identified in the JCL:

```
// DLBL UIN,'PREPROCESSOR.OUTPUT',0    *-- Output from DB2 for VSE Preprocessor
// EXTENT SYS004,.....
// ASSGN SYS004,...
// DLBL UOUT,'OBJMAINT.OUTPUT',0      *-- Output from strip operation
// EXTENT SYS005,.....
// ASSGN SYS005,...
// EXEC OBJMAINT
./ LIST PARM=JOB
./ BLOCK BLKSIZE=80
./ COPY
/*
// DLBL IJSYSIN,'OBJMAINT.OUTPUT',0    *-- Input to Translator
// EXTENT SYSIPT,.....
// ASSGN SYSIPT,..
```

Figure 46 (Part 2 of 2). Creating an Object Deck and Phase for an Online Program (Multiple User Mode)

Distributing Packages across Like and Unlike Systems

To run your application program on another DB2 Server for VSE & VM database manager, you can simply distribute its load module and the DB2 Server for VSE & VM package. (You do not have to distribute the source code and then preprocess and compile it on the other system). Reload the package to all application servers that your package accesses, and send the load module to all DB2 Server for VSE & VM application requesters that your program accesses. You can unload the package to be distributed from the application server into a file, and subsequently reload the file into the new application server. Only the owner of the package or the database administrator can unload or reload the package.

If the package is distributed among application servers that are at different release levels of the system, a run-time error occurs if the package uses a feature that is not available on the application server on which the package was reloaded. To ensure that the load module and the package that you are distributing are meant to be used together, use the preprocessor parameter CTOKEN to place the same consistency token in both the load module and the package. Refer to “Preprocessing the Program” on page 108. If the two tokens do not match, the application server stops the program from running. For information on distributing packages on both like and unlike systems, refer to the *DB2 Server for VSE & VM Database Services Utility* manual.

If your CICS/VSE application is to run at other DB2 Family application servers, do the following:

- When you preprocess your application, specify the BIND option so that a bind file is created.
- CICS translate, compile, and linkedit the preprocessor source output to create a VSE phase.

- Use the CBND transaction to generate a package at each DB2 application server at which the application is to be run. CBND uses the bind file created in the preprocessor step above.

For more information on CBND, see “Creating a Package Using CBND”.

Creating a Package Using CBND

If a bind file is generated after processing, it can be used to create a package in a remote or local application server. This enables the CICS online application program to access a remote DRDA server. To convert the SQL statements in a bind file into a package, use the CBND CICS transaction. CBND can be used to bind all applications whose bind file was generated by the preprocessor to a remote or local server. To create a package locally, for a FORTRAN application, you still need to run the batch preprocessor. Executing the preprocessor with the BIND option and executing CBND are complementary. The preprocessor creates and stores the bind file in a master or private VSAM file and CBND reads the bind file back into an application server.

If using the CBND command, you must be either the owner of the program whose bind file you are binding to a remote server or a database administrator.

A second level transaction, CB2D, is required for CBND to complete the bind process. The CB2D transaction is invoked internally by CBND through an EXEC CICS START command. One or more CB2D transactions can be started by one CBND transaction based on the number of application servers specified in the CBND DBLIST input parameter. Each CB2D is responsible for connecting to a target database and creating a package. This transaction needs to be defined during installation to fully enable the CBND function.

The CBND transaction has 13 parameters:

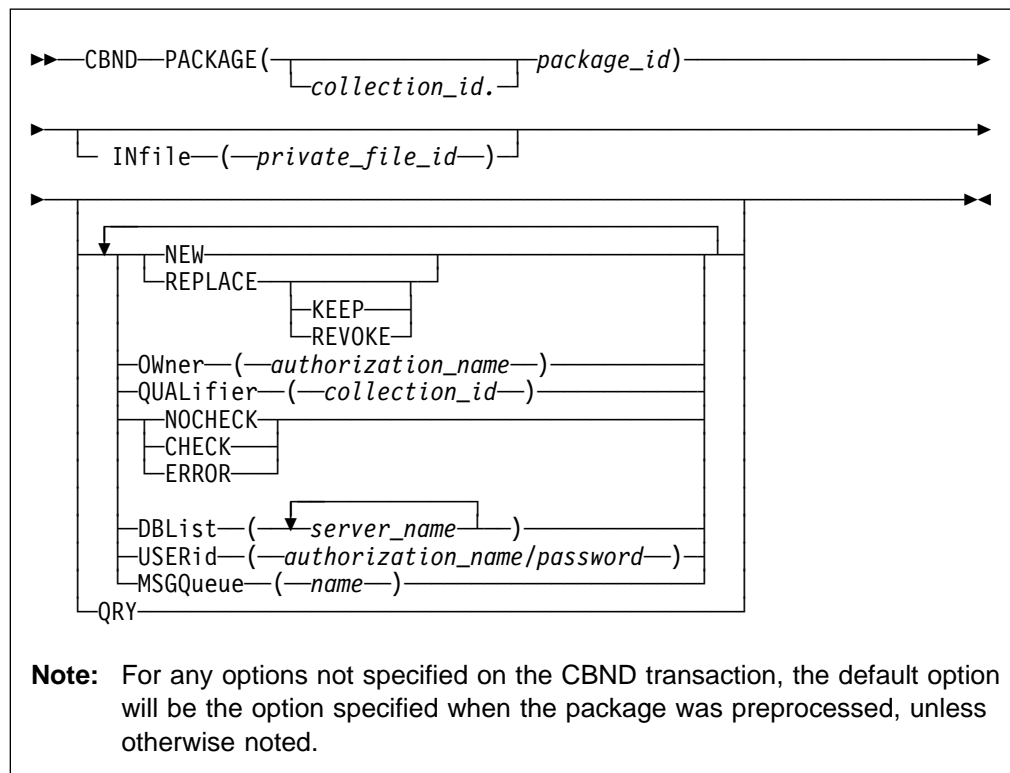


Figure 47 (Page 1 of 3). CBND Transaction Parameters

Parameter	Description
PACKAGE (positional parameter 1)	<p>This parameter identifies the package to be created and the associated bind file. The associated bind file must be created by the DB2 Server for VSE preprocessor and its contents must not be changed in any way. <i>collection_id.package_id</i> is the name by which the database manager identifies the package to be created and the associated bind file. The <i>collection_id</i> portion is optional, and fully qualifies the <i>package_id</i> and any unqualified objects referenced within the package.</p> <p>If the <i>collection_id</i> is not specified, and the <i>user_id</i> is, the <i>collection_id</i> defaults to the <i>user_id</i>. If neither is specified, the <i>collection_id</i> defaults to the connected authorization-id.</p> <p>You must be the owner of the bind file that you want to bind. To bind another user's bind file, you must have DBA authority.</p>
REPLACE	<p>This parameter is specified if an existing package is to be replaced by the bind. If the package does not exist, a new package is created without an error or warning message.</p> <p>KEEP causes the existing grants of RUN privilege to remain in effect when the package is bound. However, if the owner of the package is not entitled to grant all privileges embodied in the package, all existing grants of the RUN privilege are revoked. The KEEP and REVOKE parameters apply if the package has previously been created and the owner of the package has granted the RUN privilege on the resulting package to other users. The KEEP and REVOKE parameters are allowed only with REPLACE.</p> <p>If the REVOKE parameter is specified, all existing grants of the RUN privilege are revoked. The KEEP and REVOKE parameters are allowed only with REPLACE.</p>
NEW	<p>The NEW parameter is specified if the package to be created does not exist and is to be created. If the package with the same name and owner already exists in the remote application server, CBND fails.</p>
OWner	<p>This parameter specifies the <i>authorization_name</i> of the owner of the package being created. The OWner parameter is to be used when you are binding against a remote application server. However, if you specify this parameter when binding against a local DB2 Server for VSE & VM application server, the <i>authorization_name</i> must be the same as the application server authorization ID.</p>
QUALifier	<p>This parameter specifies the default <i>collection_id</i> within the package to resolve unqualified object names in static SQL statements.</p> <p>The QUALifier parameter is meant to be used when you are binding against a remote application server. If you specify this parameter when binding against a DB2 Server for VSE & VM application server, the <i>collection_id</i> must be the same as the application server authorization ID.</p>
NOCHECK	<p>This parameter is specified if you want the application server to check all SQL statements for validity and perform package functions. It will generate a package if no statement-parsing error was found; If you specify the NOCHECK parameter, it overrides the ERROR parameter in the bind file</p>
CHECK	<p>This parameter is specified if you want the application server to check all SQL statements for validity and generates error messages if necessary, but does not generate a package; If you specify CHECK parameter, it overrides NOCHECK or ERROR parameter in the bind file.</p>

Figure 47 (Page 2 of 3). CBND Transaction Parameters

Parameter	Description
ERROR	<p>This parameter specifies how statement parsing errors are tolerated. If the ERROR option is specified, then syntactic or semantic errors detected at the application server side will not stop the creation of the package. If the ERROR option is not specified in both CBND and the bind file, the application server will not create the package when those errors occur.</p> <p>With the ERROR option, a syntactic error will cause the DB2 for VSE & VM database to generate an Error Section in the package. At run-time, invocation of this statement will yield SQLCODE -525</p> <p>With the ERROR option, the creation of a package fails only if there was a DRDA protocol error, or a severe error detected on the AS side.</p>
QRY	<p>This parameter causes CBND to display the preprocessor options stored in the header of the bind file. The preprocessor options will be displayed at the terminal where CBND was invoked. MSGQueue is ignored if QRY is specified. No package functions will be performed if QRY is specified.</p> <p>An example of displayed output is as follows:</p> <pre> ARI5418I - THE BIND OPTIONS CURRENTLY IN EFFECT FOR - PACKAGE SQLDBA.MYPROG ARE: - NOCHECK - KEEP - BLOCK - ISQL(CS) - EXPLAIN(NO) - APPLICATION SERVER DEFAULT DATE - APPLICATION SERVER DEFAULT TIME - NOEXIST - REPLACE - APOST - CTOKEN(NO) - CCSIDS(500) - CCSIDM(0) - CCSIDG(0) - APPLICATION SERVER DEFAULT CHARSUB - DEFAULT PACKAGE LABEL </pre>
INfile	<p>This parameter identifies the 7-character VSAM file name defined to CICS through FCT or RDO (CICS/ESA R410) which contains the bind file to be loaded into the database.</p> <p>If this parameter is omitted, the bind file is assumed to reside in the master VSAM file named SQLBIND. SQLBIND is defined to CICS during installation, it is also used as the DLBL filename which identifies the bind file used for input. If your bind file resides in a private VSAM file, you will then specify the VSAM file name that matches this private VSAM file. To do this, you must make sure that this VSAM file is properly defined to CICS.</p>
DBList	<p>This parameter specifies a list of application servers on which the bind file will be bound.</p> <p>If this parameter is omitted, the bind file is bound to the default application server. For more information on establishing a default application server, see the <i>DB2 Server for VSE Database Administration</i> manual.</p>
USERid	<p>This parameter specifies the userid and password that CBND will use to execute a CONNECT statement to gain access to each application server where the package will be created.</p> <p>If this parameter is omitted, CBND will connect implicitly to the application server where the package will be created. That is, the online resource adapter will establish the userid for this particular CBND transaction. For more information on how the userid is established in the case of an implicit connect, see the <i>DB2 Server for VSE Database Administration</i> manual.</p>

Figure 47 (Page 3 of 3). CBND Transaction Parameters

Parameter	Description
MSGQueue	This parameter specifies the name of the temporary storage queue to be used to store the bind error or warning message (if any). If this parameter is omitted, no error or warning message will be stored, CBND will just display a summary report of the bind process on the terminal where CBND was invoked. The summary report contains the bind options used for the bind process, host variables, declarations, and the result of the bind. The user can browse the message queue specified using CEBR when CBND completes. This queue is kept in main storage and is not recoverable. It stays until it is purged by CEBR or if CICS is recycled. The user must determine when they should delete queues that are no longer needed to prevent possible short on temporary storage situations.

The preprocessor requires the job control statement DLBL SQLGLOB. In addition, when you want the preprocessor to generate a bind file, job control statements DLBL SQLBIND and DLBL BINDWKF are required. These DLBL statements must be provided in either the preprocess job control or the system standard label subarea. The DLBL SQLBIND indicates the VSAM file into which the preprocessor should place the output bind file. The BINDWKF indicates the work file into which the preprocessor should use when performing VSAM I/O against the bind file. The DLBL SQLGLOB contains the CHARNAME and DBCS options which are necessary for bind file creation. See Figure 48. As mentioned earlier, you can suppress SQLBIND output through CHECK and NOBIND parameters of the preprocessor.

```
// DLBL SQLBIND, 'DB2.BIND.MASTER' , ,VSAM,CAT=catalog,DISP=(OLD,KEEP)
// DLBL BINDWKF, 'DB2.BIND.WORKF' , ,VSAM,CAT=catalog
// DLBL SQLGLOB, 'DB2.SQLGLOB.MASTER' , ,VSAM,CAT=catalog,DISP=(OLD,KEEP)
```

Figure 48. DLBL Statements for SQLBIND and SQLGLOB VSAM file

A user can choose to store the bind file generated by the preprocessor in a master DB2 Server for VSE VSAM bind file, or in his or her private VSAM bind file.

If a user chose to have his application program's bind file stored in the master DB2 Server for VSE VSAM bind file, the user should specify the recommended VSAM dataset name "DB2.BIND.MASTER" on the DLBL SQLBIND job control statement. This master VSAM bind file is defined to the system when DB2 Server for VSE Version 6 Release 1 was installed. Therefore, the user does not need to define or set-up the file. However, if a different cluster name was used to define the master VSAM file, the user should then specify the corresponding dataset name used.

If a user chose to have his application program's bind file stored in a private VSAM bind file, the user must specify the name of his or her private VSAM bind file on the DLBL SQLBIND job control. The user must first define the private VSAM bind file, using the same VSAM characteristics as the master VSAM bind file. For more information on how to define the VSAM cluster for the private bind file, see *DB2 Server for VSE Installation* manual.

Chapter 5. Testing and Debugging

Contents

Doing Your Own Testing	146
Checking Warnings and Errors at Preprocessor Time	146
Testing SQL Statements	147
Using the Automatic Error-Handling Facilities	147
Using the SQLCA	148
Using the SQLERRM Field	149
Using the SQLWARN Field	150
Examining Errors	150
Using the WHENEVER Statement	150
Determining the Scope of the WHENEVER Statement	151
Examining the SQLCA	151
Using TEXT Files to Get SQLCA Field Information	153
Handling Errors in a Select-List	155
Handling Arithmetic Errors	155
Handling Numeric Conversion Errors	156
Handling CCSID Conversion Errors	157

Doing Your Own Testing

Checking Warnings and Errors at Preprocessor Time

If errors or warnings are detected during preprocessing, the preprocessor inserts messages into the modified source code and preprocessor listing files to indicate that a problem was encountered. A return code is also issued indicating the severity of the problem.

Messages associated with warning conditions are inserted into the files in comment form so that the compilation of the modified source code is not inhibited. A return code of 4 is issued and, if no problems of greater severity are found, package processing occurs. All warnings should be investigated, because they may indicate a situation that must be corrected before the program is executed. For example, if you use the NOEXIST preprocessor option and a table is not found at preprocessing time, the database manager issues a warning because it assumes that the table will be created before the program is executed. If the program is executed and the table is not found, the successful execution of the program is inhibited and an error results.

When you are preprocessing, messages associated with error conditions are inserted in the files in uncommented form so that the compilation process is inhibited. A return code of 8 or greater is issued, and no package processing occurs.

To check for error and warning conditions:

1. Scan the modified source code or preprocessor listing file for error and warning message. Error message numbers are formatted as ARInnnnE, and warning message numbers as ARInnnnW or ARInnnnI.

Messages generated during preprocessor initialization and termination are stored in the listing file, not in the modified source code. A listing file containing these messages is produced regardless of whether the NOPRint preprocessor parameter is specified.

2. Look up the message numbers, SQLCODES and SQLSTATES issued by the system in the *DB2 Server for VSE Messages and Codes* manual.
3. If the error was detected by a non-DB2 Server for VSE application server, look up the SQLCODE and SQLSTATE explanations in the *Messages and Codes* manual associated with the database management system that detected the problem.
4. Use ISQL HELP for online information about messages and SQLCODEs. To obtain limited online help information for SQLSTATEs, type HELP SQLSTATE from ISQL.

Testing SQL Statements

Several facilities are available to help you test SQL statements:

- Interactive facilities such as ISQL or QMF can be used for testing statements; however, the range of statements that can be tested with these facilities is limited. For example, you cannot test the following:
 - Statements that use host language delimiters, host variables, cursors, or statement names.
 - Dynamic or extended dynamic SQL statements.

For other static SQL statements, however, these facilities are fast and easy to use, and you can do data definitions, authorizations, and data control tasks. For information on ISQL, see the *DB2 Server for VSE & VM Interactive SQL Guide and Reference* manual.

- The DBS Utility can be used for testing the same range of SQL statements that can be tested using ISQL or QMF. In addition, it lets you use file input and output, which makes submitting and reviewing test conditions easier, and lets you set up and restore test databases with DBS Utility data load and unload commands. For information on the DBS Utility, see the *DB2 Server for VSE & VM Database Services Utility* manual.

Using the Automatic Error-Handling Facilities

Every SQL application program must provide for error handling, by declaring an SQL Communications Area (or alternatively, just the SQLCODE variable, as described later in this section). This area receives messages that the database manager sends to the program. By testing certain fields of this area, you can test for certain conditions during the program's execution.

Error handling helps protect the integrity of the database when a program fails. For example, consider the two-step operation needed to transfer \$500 from one account to another in a bank:

1. Subtract \$500 from account A
2. Add \$500 to account B.

If the system or your program fails after the first statement is executed, some customer has just “lost” \$500. This type of incomplete update is said to leave the database in an *inconsistent state*.

To avoid creating an inconsistent state, use a logical unit of work (LUW). An LUW is a group of related SQL statements, possibly with intervening host language code, that you want treated as a unit. The two steps in the previous example would make up a single LUW.

LUWs prevent inconsistencies caused by system errors or SQL statement errors. For system errors, the system automatically restores all changes made during the LUW where it encountered the error. This rollback of the LUW is identified by a negative SQLCODE and a W in the SQLWARN6. When a non-severe SQL error occurs, the system restores all changes made by the statement in error. This statement rollback is identified by a negative SQL code and a blank in SQLWARN6. For work done in the LUW before execution of the statement in error, do the following:

- Declare an SQL Communications Area (or just the SQLCODE variable)
- Code an SQL WHENEVER statement
- Code the actions to be taken if an error occurs.

To declare the SQL Communications Area (SQLCA), code this statement in your program:

```
INCLUDE SQLCA
```

When you preprocess your program, the system inserts host language variable declarations in place of the INCLUDE SQLCA statement, and SQL communicates with your program using this group of variables. The system uses the variables for warning flags, error codes, and diagnostic information. All these variables are discussed in the *DB2 Server for VSE & VM SQL Reference* manual.

The system returns a return code in SQLCODE after executing each SQL statement. When a statement is executed successfully, SQLCODE is set to 0 (SQLSTATE is '00000'). The system indicates error conditions by returning a negative SQLCODE. A positive SQLCODE indicates normal or warning conditions experienced while executing the statement.

The system also returns a return code in SQLSTATE after executing each SQL statement. SQLSTATE provides common return codes for IBM's relational database products. SQLSTATE values comply with the SQL92 standard. For a discussion of return codes in SQLSTATE, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

The system supports a stand-alone SQLCODE. If you request this support, you must not include the SQLCA definition in your program. You must, however, provide the integer variable SQLCODE (SQLCOD in FORTRAN). Refer to "Defining the Preprocessing Parameters" on page 112 for information on the preprocessor parameters that provide NOSQLCA support.

The following WHENEVER statement specifies a system action that is to occur when an SQL error (that is, a negative SQLCODE) is returned:

```
WHENEVER SQLERROR GO TO ERRCHK
```

That is, whenever an SQL error (SQLERROR) occurs, program control is transferred to code which follows a specific label, such as ERRCHK. This code should include logic to analyze the error indicators in the SQLCA. Depending on how ERRCHK is defined, action may be taken to execute the next sequential program instruction, to carry out some special functions, or, as in most cases, to roll back the current LUW and end the program.

Using the SQLCA

As mentioned previously, the database manager returns a return code in the SQLCA after almost every SQL statement. The only statements that do not return SQLCODEs are SQL declarative statements, which are not executed; therefore, no SQLCODE can be returned. (Never test for an SQLCODE after a declarative statement.) The following are examples of declarative statements:

- BEGIN DECLARE SECTION
- END DECLARE SECTION
- WHENEVER
- INCLUDE SQLCA
- INCLUDE SQLDA.

When a nondeclarative statement is in error, the system reverses any changes to the database caused by that statement. For any previous work done in the LUW, you have to tell the system what action to take.

Figure 49 shows a representation of the SQLCA structure with host-language independent data type descriptions. (Refer to the appendixes for the SQLCA data types of a particular programming language.)

Figure 49. SQLCA Structure (in Pseudocode)

```
SQLCA -- a structure composed of:
  SQLCAID -- character string of length 8
  SQLCABC -- 31-bit binary integer
  SQLCODE -- 31-bit binary integer
  SQLERRM -- varying character string of maximum length 70
  SQLERRP -- character string of length 8
  SQLERRD -- an array composed of:
    SQLERRD(1) -- 31-bit binary integer
    SQLERRD(2) -- 31-bit binary integer
    SQLERRD(3) -- 31-bit binary integer
    SQLERRD(4) -- 31-bit binary integer
    SQLERRD(5) -- 31-bit binary integer
    SQLERRD(6) -- 31-bit binary integer
  SQLWARN -- a sub-structure composed of:
    SQLWARN0 -- single character
    SQLWARN1 -- single character
    SQLWARN2 -- single character
    SQLWARN3 -- single character
    SQLWARN4 -- single character
    SQLWARN5 -- single character
    SQLWARN6 -- single character
    SQLWARN7 -- single character
    SQLWARN8 -- single character
    SQLWARN9 -- single character
    SQLWARNA -- single character
  SQLSTATE - character string of length 5
```

The *DB2 Server for VSE & VM SQL Reference* manual explains the structure of the SQLCA, and describes each field in detail. Some tips about SQLERRM and SQLWARN fields are provided below.

Using the SQLERRM Field

The message texts associated with particular SQLCODEs (which can be found in the *DB2 Server for VSE Messages and Codes* manual), often include variables which are returned in the SQLERRM field of the SQLCA. In some situations, the format of the last variable in the SQLERRM field is 'FOnn', which specifies the format number of the SQLCODE message text. The 'FO' is an abbreviation for *format*, and 'nn' represents the number that identifies the version of the message text that applies. If there is more than one variable returned through SQLERRM, the variables are separated by X'FF'.

The first two bytes of SQLERRM (which is varying-length) contain the total length of the string.

See "Handling Numeric Conversion Errors" on page 156 for the values of this field when a numeric conversion occurs in an outer select and "Handling Errors in a

Select-List” on page 155 for the values when an error occurs while evaluating expressions in an outer select.

Using the SQLWARN Field

This field contains characters that warn of various conditions encountered during the processing of your statement. Alternatively, specific warnings may be indicated by positive values in the SQLCA field, SQLCODE. For example, a warning indicator is set when the system ignores null values in computing an average. When the system encounters a particular condition, it sets the corresponding warning character to a designated value, such as W, N, or Z. When the system encounters two different warning conditions and must set the warning character to either W or N, the system randomly chooses one value. If the system encounters three different warning conditions and must set the value of the warning character to W, N, or Z, the system sets the value of the warning character to W or N, but not Z. The warning character Z is, therefore, overridden by W or N. One or more warning characters may be set to W regardless of the code returned in SQLCODE. The meanings of the warning characters are listed in the *DB2 Server for VSE & VM SQL Reference* manual.

Because there is only one return code structure in each program, you should copy out of the structure any information that you wish to save before the next SQL statement is executed. Of particular note are the SQLCODE and the warning indicators (SQLWARN).

Examining Errors

Using the WHENEVER Statement

The WHENEVER statement is a nonexecutable statement that assists you in reacting to unusual conditions, based on data returned in the SQLCA.

The following three conditions can be addressed with WHENEVER statements:

SQLERROR Occurs when SQLCODE is negative.

SQLWARNING Occurs when SQLCODE is positive but not 100, or when SQLCODE is zero and SQLWARN0 is W.

NOT FOUND Occurs when SQLCODE is 100 (SQLSTATE is '02000').

Each SQL statement is within the scope of one WHENEVER statement for each of the three conditions. A WHENEVER statement for an already specified condition can be overridden at any time by coding another WHENEVER statement for the same conditions.

One of three actions can be taken for a WHENEVER statement:

GOTO or GO TO Transfers control to a specified location.

STOP Terminates the program. The STOP action cannot be used with the NOT FOUND condition.

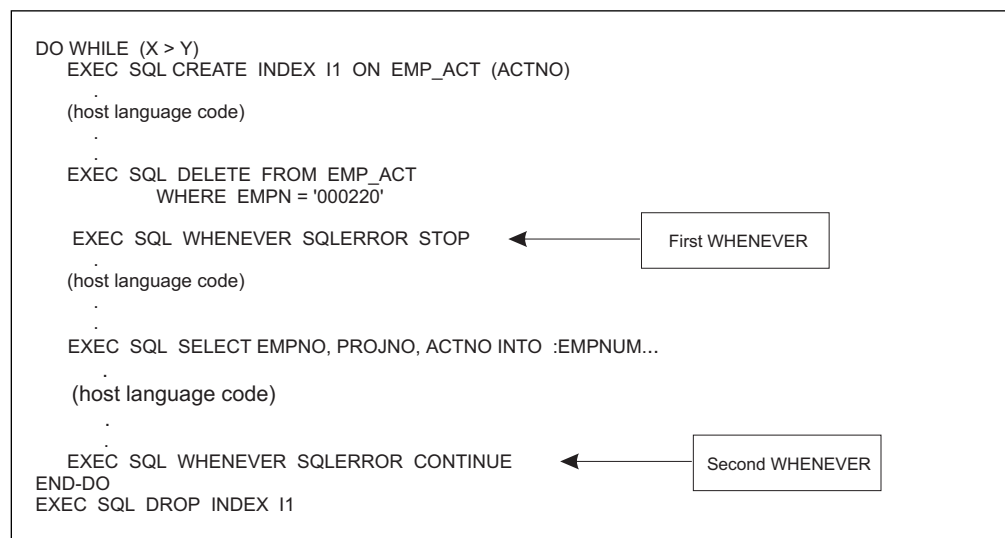
CONTINUE Executes the next sequential instruction.

If a WHENEVER statement is not coded for a condition, it is processed as if the condition were CONTINUE.

For a full discussion of the WHENEVER statement, see the *DB2 Server for VSE & VM SQL Reference* manual.

Determining the Scope of the WHENEVER Statement

The scope of a WHENEVER statement is determined by its position in the source program listing, not by its place in the logic flow. (This is because WHENEVER is a declarative statement.) For example:



In the pseudocode program fragment above, the scope of the first WHENEVER is only the SELECT INTO statement. The second WHENEVER applies to the DROP INDEX statement (and to all SQL statements that follow it until another WHENEVER is encountered). The CREATE INDEX and DELETE statements are not covered by a WHENEVER (there is no preceding WHENEVER); therefore, the default CONTINUE action applies for WHENEVER conditions.

Examining the SQLCA

The SQLCA structure can be examined using the WHENEVER statement. You can test for both general (SQLCODE < 0 | SQLWARN0 <> blank) and specific (SQLCODE = -911 | SQLWARN6 = 'W') warning or error conditions. To do this, use a WHENEVER statement with a GOTO somewhere in the source program before the SQL statements for which you want to directly examine the SQLCA.

For example, Figure 50 on page 152 shows pseudocode for an error handling routine:

```

EXEC SQL WHENEVER SQLERROR GOTO ERRCHK
.
.
.
ERRCHK: * Prevent further errors from branching here
EXEC SQL WHENEVER SQLERROR CONTINUE
* Handle severe errors first
IF SQLWARN0 = 'S'
    DISPLAY('A SEVERE ERROR HAS OCCURRED.')
    DISPLAY('SQLCODE = ' SQLCODE)
.
.
.
STOP
END-IF
* Describe the error
DISPLAY('AN ERROR HAS OCCURRED.')
DISPLAY('SQLCODE = ' SQLCODE)
.
.
.
EXEC SQL ROLLBACK WORK
* Check for errors
IF SQLCODE < 0
    DISPLAY('ROLLBACK WORK FAILED. SQLCODE = ')
    DISPLAY(SQLCODE)
.
.
.
* Recovery from error is complete.
ELSE
    DISPLAY('ROLLBACK WORK SUCCEEDED.')
.
.
.
END-IF

```

Figure 50. Pseudocode Error-Handling Routine

When an error occurs, control is passed to the ERRCHK label. Then, in order to prevent a program loop in this routine, a WHENEVER SQLERROR CONTINUE statement is issued. (It is safe to do this because WHENEVER statements never return an SQLCODE.) Next, the severity of the error is determined. If a severe error occurs, the execution of any SQL statements on this application server (except a CONNECT statement) terminates the application abnormally. The pseudocode example reports the error and ends.

If the error is not severe, the pseudocode example displays an informational message giving the SQLCODE, and an attempt is made to undo any changes. The pseudocode example determines whether the ROLLBACK successfully completed, by checking the SQLCODE after the ROLLBACK statement.

After a severe error, only a CONNECT statement is permitted. If the application program reconnects to the application server in which the severe error occurred, two possibilities exist. If the application server has been restarted or has otherwise recovered, the application may continue; otherwise, another severe error will result.

If your application program is accessing multiple application servers, you can enter a CONNECT statement to switch to another application server and continue processing.

Using TEXT Files to Get SQLCA Field Information: When an SQL error occurs, you can examine the SQLCA in order to determine the problem. To reduce the time taken to do so, you can issue a call from the application to either module module

```
CALL ARISSMA, (SQLCA, S1, S2, S3, S4, S5), VL      /* Assembly Language */
ARISSMA(SQLCA, S1, S2, S3, S4, S5)                /* 'C' */
CALL 'ARISSMA' USING SQLCA S1 S2 S3 S4 S5.        /* COBOL */
CALL ARISSMF(SQLCA, SQLERP, S1, S2, S3, S4, S5)   /* FORTRAN */
CALL ARISSMA(SQLCA, S1, S2, S3, S4, S5);         /* PL/I */
```

In this example, S1, S2, S3, S4, and S5 are character strings declared within the program and according to the rules of the specific language. Each string will contain information on specific SQLCA fields, after the call to ARISSMA/ARISSMF. Figure 51 shows the parameter name for the strings, their SQL name, their lengths, and the corresponding SQLCA fields.

<i>Figure 51. SQLCA Error Information Strings</i>			
Parameter Name	SQL Name	Length	SQLCA Field
S1	SQLCSTR1	13	SQLCODE
S2	SQLCSTR2	13	SQLERRD1
S3	SQLCSTR3	13	SQLERRD2
S4	SQLCSTR4	12	SQLERRP (part 1)
S5	SQLCSTR5	14	SQLERRP (part 2)

Notes:

1. For assembler you must INCLUDE modules ARIPRDID and ARISSMA before the link-edit step.
2. For C you must:
 - Declare the strings 1 character longer than that shown in the table. Also, within the program itself, you must append the end-of-string character “\0” to the last position within each of the character strings before displaying them on the screen.
 - Include the statement

```
#pragma linkage (ARISSMA,OS);
```

to indicate that System/390 linkage is used in the call to ARISSMA.
 - INCLUDE modules ARIPRDID and ARISSMA before the link-edit step.
3. For COBOL you must INCLUDE modules ARIPRDID, ARIPADR4, and ARISSMA before the link-edit step. If the program was preprocessed prior to SQL/DS Version 2 Release 2, use ARIPADR instead of ARIPADR4.
4. For FORTRAN you must INCLUDE modules ARIPRDID, ARIPEIFA, ARISSMF, and ARIPSTR before the link-edit step.
5. For PL/I you must:

- Declare ARISSMA as an external entry point to indicate that System/390 linkage is used in the call to ARISSMA:

```
DCL ARISSMA ENTRY EXTERNAL OPTIONS(ASM,RETCODE);
```

- INCLUDE modules ARIPRDID and ARISSMA before the link-edit step.

ARISSMA/ARISSMF returns information in the strings to your program. This information can be displayed or can be written to a file. The format in which the information is returned is shown below.

- SQLCSTR1** PRCS/*nnnnnnnn*; where *n* is the decimal representation of the absolute value of the SQLCODE, right-justified, and padded with 0's for a total length of 8 digits.
- SQLCSTR2** PRCS/*nnnnnnnn*; where *n* is the decimal representation of the absolute value of the SQLERRD1, right-justified and padded with 0's for a total length of 8 digits.
- SQLCSTR3** PRCS/*nnnnnnnn*; where *n* is the decimal representation of the absolute value of the SQLERRD2, right-justified and padded with 0's for a total length of 8 digits.
- SQLCSTR4** FLDS/SQLERRP. This value is always returned in the string.
- SQLCSTR5** VALU/*Caaaaaaaa*; where *a* is left-justified, padded by blanks, and is the module name provided in field SQLERRP.

Suppose the SQLCA fields have the following values when the error occurred:

```
SQLCODE = -901
SQLERRD1 = -160
SQLERRD2 = -33
SQLERRP = ARIXOEX
```

then the values of the strings will be:

```
SQLCSTR1 ==> PRCS/00000901
SQLCSTR2 ==> PRCS/00000160
SQLCSTR3 ==> PRCS/00000033
SQLCSTR4 ==> FLDS/SQLERRP
SQLCSTR5 ==> VALU/CARIXOEX
```

These values may be displayed as shown in the pseudocode in Figure 52:

```
EXEC SQL WHENEVER SQLERROR GOTO ERROR
.
.
.
ERROR: * Display string information
      CALL ARISSMA
      DISPLAY('SQLCSTR1 ='SQLCSTR1)
      DISPLAY('SQLCSTR2 ='SQLCSTR2)
      DISPLAY('SQLCSTR3 ='SQLCSTR3)
      DISPLAY('SQLCSTR4 ='SQLCSTR4)
      DISPLAY('SQLCSTR5 ='SQLCSTR5)
```

Figure 52. Pseudocode to Display Error Information

Another reason an application might want to process the SQLERROR condition is for graceful cleanup and termination. An example of this is a CICS/VSE

conversational transaction. Rather than abnormally terminating on SQLERROR, a conversational transaction might do the following:

- Issue a CICS/VSE SYNCPOINT ROLLBACK or SQL ROLLBACK. This backs out uncommitted changes to CICS/VSE and DB2 Server for VSE recoverable resources.
- Issue an error message to the terminal user. Such a message typically informs the user of the state of affairs and identifies the user's options for proceeding with the transaction.

The ISQL transaction handles SQLERROR in a similar manner. That is, rather than terminating the ISQL session, the user is given an error message and allowed to proceed.

Handling Errors in a Select-List

The database manager tolerates the occurrence of certain errors resulting from the execution of expressions occurring in a *select_list* of an outer select statement.

Handling Arithmetic Errors

The arithmetic errors that can be tolerated are listed in Figure 53.

Arithmetic Errors That Will Be Tolerated
<ul style="list-style-type: none">• DECIMAL<ul style="list-style-type: none">– Divide Exception– Decimal Overflow• FLOAT<ul style="list-style-type: none">– Divide Exception– Exponent Overflow• INTEGER, SMALLINT<ul style="list-style-type: none">– Divide Exception– Fixed Point Overflow

Note: FLOAT can be either single-precision or double-precision float. Refer to “Assigning Data Types When the Column Is Created” on page 43 for more information on floating-point data types.

Errors in date and time arithmetic are not tolerated. For example,

```
DATE('9999-12-31') + 1 DAY
```

results in a negative SQLCODE because the result would be an invalid date.

In the next example however, if the value of C2 is zero, the arithmetic error would be tolerated.

```
DATE('1996-12-31') + C1/C2 DAYS
```

The expression in the outer *select_list* may be by itself, or it can be an argument in a scalar or column function other than the column functions AVG and SUM.

If the errors occur on an outer *select_list*, and every output host variable that is associated with the expression that is in error has an associated output indicator variable, the system does the following:

- The output indicator variable for each arithmetic expression in error is set to -2.
- A positive warning SQLCODE is placed in the SQLCA.
- SQLWARN0 in the SQLCA remains unaffected.
- The value of the associated host variable is undefined.
- Execution of the statement continues, such that all expressions and values not having arithmetic errors are returned.
- If the statement is a FETCH, the cursor remains open.

However, if the errors do not occur on the outer *select_list*, or if there are arithmetic errors on the *select_list* and not every output host variable that is associated with the expression in error has an associated output indicator variable, the system takes the following actions:

- A negative error SQLCODE -802 (SQLSTATE '22003') is returned in the SQLCA.
- The values of the host variables and any supplied indicator variables in the *select_list* are undefined.
- Execution of the statement is halted.
- If the statement is a FETCH, the cursor will remain open.

In either case, the SQLERRM of the SQLCA error message will identify the first expression in error in the outer *select_list*. The following are returned in the error message:

- The exception type
- The arithmetic operation being performed at the time of the error
- The data type of the *select_list* items being manipulated
- The ordinal position of the expression in error.

Depending on when the error is detected, some parts of the error message will be blank.

Handling Numeric Conversion Errors

The numeric conversion errors that can be tolerated are listed in Figure 54.

<i>Figure 54. Tolerated Numeric Conversion Errors</i>
<ul style="list-style-type: none">• FLOAT to<ul style="list-style-type: none">– DECIMAL– INTEGER– SMALLINT• DECIMAL to<ul style="list-style-type: none">– DECIMAL– FLOAT– INTEGER– SMALLINT• INTEGER to<ul style="list-style-type: none">– DECIMAL– SMALLINT• SMALLINT to<ul style="list-style-type: none">– DECIMAL

Note: FLOAT can be either single-precision or double-precision float. Refer to “Assigning Data Types When the Column Is Created” on page 43 for more information on floating-point data types.

If an error occurs while converting numeric values into the data type of the host variables, and output indicator variables are provided with host variables, for which numeric conversion errors occurred, the system does the following:

- The output indicator variable for each host variable for which a numeric conversion error occurred is set to -2.
- If no other warning SQLCODE is contained in the SQLCA, then a positive warning SQLCODE is placed in the SQLCA and the error message tokens in SQLERRM will identify the first conversion error.
- SQLWARN0 in the SQLCA is unaffected.
- The values of the associated host variables are undefined.
- Execution of the statement continues such that all values not in error are returned to your program.
- If the statement is a FETCH then the cursor will remain open.

If output indicator variables are *not* provided for host variables for which numeric conversion errors occurred, the system does the following:

- A negative error SQLCODE is placed in the SQLCA
- The SQLCA error message tokens identify the first conversion error
- The values of the host variables and indicators are undefined
- Execution of the statement is halted
- If the statement is a FETCH then the cursor will remain open.

In either case, the SQLERRM of the SQLCA will identify the first expression in error, and the following will be returned in the error message:

1. The data type of the value being moved into the host variable
2. The ordinal position of the expression in error
3. The data type of the host variable.

Handling CCSID Conversion Errors

The database manager tolerates CCSID conversion errors in which a character or characters have been mapped to the defined error byte.

If this occurred during CCSID conversion of data to be returned to the user, and output indicator variables *are* provided with host variables, the system does the following:

- For each host variable for which the CCSID conversion error byte mapping has occurred, the output indicator is set to -2.
- If no other warning SQLCODE is in the SQLCA, a positive warning SQLCODE is placed there, and the error message tokens in SQLERRM will identify the first conversion error.
- SQLWARN0 in the SQLCA is unaffected.
- The values of the associated host variables are undefined.
- Execution of the statement continues and returns all correct values to your program.

- For a FETCH statement, the cursor remains open.

If output indicator variables are *not* provided for host variables for which CCSID conversion errors occurred, the database manager does the following:

- A negative SQLCODE is placed in the SQLCA.
- The SQLCA error message tokens identify the first conversion error.
- The values of the host variables, and any indicators, are undefined.
- Execution of the statement is halted.
- For a FETCH statement, the cursor remains open.

Chapter 6. Using Dynamic Statements

Contents

Dynamically Defining SQL Statements	160
Comparing Non-Query Statements to Query Statements	160
Using Non-Query Statements	160
Executing Non-Parameterized Statements	161
Executing Parameterized Statements	162
Using Query Statements	164
Executing a Non-Parameterized Select-Statement	164
Using the PREPARE and DESCRIBE Statements	164
Declaring the SQL Descriptor Area (SQLDA)	165
Processing a Run-Time Query Using the SQLDA	165
Allocating Storage for the SQLDA Using the SQLVAR Array	166
Initializing the SQLN Field of the SQLDA	166
Inserting Values in the SQLDA	167
Analyzing the Elements of SQLVAR	168
Allocating Storage for the Result of the Select-Statement	169
Retrieving the Query Result	171
Executing a Parameterized SELECT Statement	171
Generating an Additional SELECT Statement	171
Executing a Parameterized Non-Query Statement	175
Generating a SELECT Statement	175
Using an Alternative to a Scanning Routine	176
Ensuring Data Type Equivalence in a Dynamically Defined Query	178
Summarizing the Fields of the SQLDA	179
Using the SQLN Field	180
Using the SQLD Field in the SQLDA	180
Using the PREPARE Statement	181
SQL Functions Not Supported in Dynamic Statements	182

Dynamically Defining SQL Statements

Previous chapters have described how to code various SQL statements directly into a program and have the database manager preprocess them. For some kinds of applications, however, it is desirable to execute SQL statements that are not known until the program is actually running. An example would be a program to support an interactive user who wishes to type queries and receive results at a terminal. In this case, you cannot embed the SQL statements in the program and have the DB2 Server for VSE preprocessor recognize them, because the program reads the statements from a terminal when it is running. To support applications such as this, the system provides facilities for executing SQL statements that are specified at run time.

For a detailed description of each of these statements, see the *DB2 Server for VSE & VM SQL Reference* manual. The following SQL statements define dynamic statements.

- PREPARE - prepares a single statement for execution
- DESCRIBE - obtains information about columns in the *select_list* of a prepared *select-statement*
- EXECUTE - executes a non-*select-statement* in a package
- EXECUTE IMMEDIATE - prepares a single statement and immediately executes it
- DECLARE CURSOR - in connection with OPEN, FETCH, PUT, and CLOSE, executes a SELECT or an INSERT statement
- OPEN (cursor)
- FETCH (cursor)
- PUT (cursor)
- CLOSE (cursor).

Comparing Non-Query Statements to Query Statements

The SQL statements that you can dynamically define and execute fall into one of two categories: non-query SQL statements (such as ALTER, CREATE, DELETE, INSERT, and UPDATE) and query statements (such as SELECT). General usage techniques for both categories are discussed below; specific statement syntax is shown in the following sections.

Using Non-Query Statements

Executing Non-Parameterized Statements

The simplest SQL statements to execute dynamically are those that do not return any result other than values in the SQLCA. No output host variables are used. This is the case with all data definition and data control statements, and with all data manipulation statements except SELECT.

Suppose an inventory control program is designed around the following table:

```
CREATE TABLE INVENTORY
(PARTNO          SMALLINT          NOT NULL,
DESCRIPTION     VARCHAR(24)        ,
QONHAND        INTEGER            )
```

The program reads SQL DELETE statements similar to these from a terminal:

```
DELETE FROM INVENTORY WHERE PARTNO =221
DELETE FROM INVENTORY WHERE PARTNO =315
DELETE FROM INVENTORY WHERE PARTNO =807
```

After reading a statement, the program immediately executes it.

SQL statements must be prepared before they can be executed. Because the SQL statements are read at run time, they have not been prepared. An SQL statement called EXECUTE IMMEDIATE causes an SQL statement to be prepared and executed—all at run time. Here is a pseudocode solution to the above problem:

```
EXEC SQL BEGIN DECLARE SECTION
      DECLARE DSTRING VARYING CHARACTER (80)
      .
      .
EXEC SQL END DECLARE SECTION

READ DSTRING FROM TERMINAL
EXEC SQL EXECUTE IMMEDIATE :DSTRING
```

A DELETE statement is read into a host variable called DSTRING. DSTRING is then used as a parameter in the EXECUTE IMMEDIATE statement, causing the DELETE statement to be immediately prepared and executed.

A host variable can be used as a parameter for the EXECUTE IMMEDIATE statement. The table below shows how the host variable must be declared in the different languages:

Language	Fixed-Length Variable	Varying-Length Variable	String Constant
Assembler		X	
C		X	
COBOL		X	X
FORTRAN	X		X
PL/I	X	X	X

The *Fixed-Length Variable* refers to CHAR host variables, *Varying-Length Variable* refers to VARCHAR host variables, and *String Constant* refers to quoted character

string constants. The following is an example of a *String Constant* dynamic statement:

```
EXECUTE IMMEDIATE 'DELETE FROM INVENTORY WHERE PARTNO=201'
```

The SQL statement submitted to EXECUTE IMMEDIATE must *not* contain host language delimiters or SQL delimited identifiers. That is, the statement must be in basic form. Avoid using either delimited identifiers or strings of DBCS characters in statements specified in string constants.

Note: The preferred method is to use a host variable rather than the string constant.

The EXECUTE IMMEDIATE statement itself, however, must have appropriate delimiters. For example, in COBOL all SQL statements must be preceded by EXEC SQL, and followed by the END-EXEC keyword as follows:

```
EXEC SQL EXECUTE IMMEDIATE
    'DELETE FROM INVENTORY WHERE PARTNO = 201'
END-EXEC.
```

If the host language you are using permits it, you can concatenate a constant to a variable. For example, PL/I uses two vertical bars (||) as the concatenation symbol:

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM INVENTORY WHERE' || PREDS;
```

Note: The concatenation symbol used here is *not* the same as the concatenation operator discussed in Chapter 3, “Coding the Body of a Program” on page 21.

The “EXEC SQL” and the semicolon (;) are the host language delimiters for PL/I. At run time, the variable PREDS should contain a character string representing one or more predicates that complete the DELETE statement. The variable PREDS must not be used as a host variable, since it is being concatenated to the constant string.

Executing Parameterized Statements

In the example above, note that the DELETE statements that were dynamically executed contained no host variables. That is, they were executed only once, with a single value for PARTNO. Suppose that you wanted to execute the DELETE statement repeatedly with different values, without having to key in the entire statement each time. Consider how it might be done if you coded the DELETE statement directly in a program:

```
READ PART FROM SYSIPT
DO WHILE (PART <= 0)
    EXEC SQL DELETE FROM INVENTORY WHERE PARTNO = :PART
    READ PART FROM SYSIPT
END-DO
```

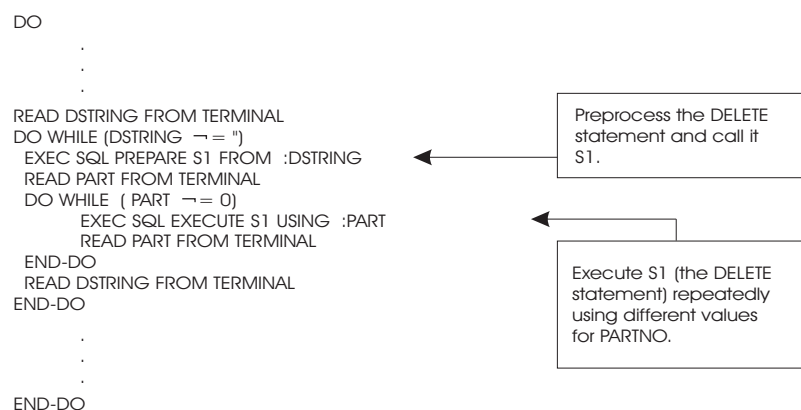
The loop is repeated until a PART of 0 is read.

Now, suppose that you wish to read both the DELETE statement and the part numbers from a terminal for dynamic execution. When this is done, the DELETE statement itself should not contain host variables; rather, it should contain question marks (?) to indicate where the value is to be substituted:

```
DELETE FROM INVENTORY WHERE PARTNO = ?
```

This type of statement is called a *parameterized* SQL statement (a *parameter* is an input host variable). Thus far, none of the dynamic statements contained any parameter markers, and they could be executed using EXECUTE IMMEDIATE. Parameterized SQL statements require a slightly more complex facility called PREPARE and EXECUTE. This facility can be thought of as an EXECUTE

IMMEDIATE performed in two steps. The first step (PREPARE) causes the parameterized statement to be prepared, and gives it a name of your choosing. (This name should not be declared as a host variable.) The second step (EXECUTE) causes the statement to be executed using values that you supply for the parameters. After a statement is prepared, it can be executed many times. Here is the pseudocode:



You must not execute a dynamically defined statement after ending the logical unit of work in which the statement was prepared. If you do, an error is issued.

In routines similar to the above example, the number of parameters and their data types must be known, because the host variables that provide input data are declared when the program is being written.

Naturally, this greatly limits the number of different SQL statements that you can read in. In the above example, the only SQL statements that can be executed are those containing a single parameter. This single parameter is defined as a 15-bit integer in the program, and must be used as such. For example, the pseudocode above can also process the statements below. (At the terminal, the user types in a statement followed by values for the parameter markers.)

```
INSERT INTO INVENTORY (PARTNO) VALUES(?)
```

For each value you provide for “?”, the INSERT statement is executed, and a new row is inserted into INVENTORY. The value you provide is placed in the PARTNO column. The other columns of the table are given the null value (provided they are nullable).

```
UPDATE INVENTORY SET DESCRIPTION = 'GEAR' WHERE PARTNO = ?
```

For each value you provide for “?”, the UPDATE statement is executed, and the DESCRIPTION column of the INVENTORY table is set to 'GEAR'.

```
UPDATE INVENTORY SET QONHAND = 0 WHERE PARTNO = ?
```

For each value you provide for “?”, the UPDATE statement is executed, and the QONHAND column in the INVENTORY table is set to 0.

Obviously there are some applications for this kind of dynamic statement processing, but they are quite specialized. Suppose new parts are added to the inventory. Each part is a different kind of gear, and none of the parts are yet in the warehouse. The input stream for the pseudocode above would be as follows:

```
INSERT INTO INVENTORY (PARTNO) VALUES (?)
301
302
303
304
0
UPDATE INVENTORY SET DESCRIPTION = 'GEAR' WHERE PARTNO = ?
301
302
303
304
0
UPDATE INVENTORY SET QONHAND = 0 WHERE PARTNO = ?
301
302
303
304
0
```

Using Query Statements

Executing a Non-Parameterized Select-Statement

Using the PREPARE and DESCRIBE Statements

A somewhat more complex facility is needed for executing a dynamically defined *select-statement*. Usually, a select-statement returns the result of a query into one or more host variables. When the query is read from a terminal at run time, however, you cannot know in advance how many and what type of variables to allocate to receive the result. The database manager therefore provides a special statement called DESCRIBE by which a program can obtain a description of the data types of a query result. After using the DESCRIBE statement, the program can dynamically allocate storage areas of the correct size and type to receive the result of the query. If DESCRIBE is used on a prepared SQL statement that was not a SELECT, the system indicates this by returning a zero in the variable SQLD of the SQL descriptor area.

When handling a run-time query, the program first uses the PREPARE statement which (as in the previous section) preprocesses the SQL statement. The PREPARE step also associates a statement-name with the query. The DESCRIBE statement is then used to obtain a description of the answer set. On the basis of this description, the program dynamically allocates a storage area suitable to hold one row of the result. The program then reads the query result by associating the name of the statement with a cursor and by using cursor manipulation statements (OPEN, FETCH, and CLOSE).

SELECT INTO statements cannot be executed dynamically.

Declaring the SQL Descriptor Area (SQLDA)

Dynamically defined queries center around a structure called the SQL Descriptor Area (SQLDA). The SQLDA is usually a based structure; that is, storage for it is allocated dynamically at run time. Figure 55 is a representation of the SQLDA structure with host-language-independent data type descriptions. Each host language has different considerations for the SQLDA structure; you should read the section on dynamic statements in the appropriate appendix before you attempt to code a program that uses the SQLDA. In addition, see “Summarizing the Fields of the SQLDA” on page 179 for information about the fields of the SQLDA.

```
SQLDA -- a based structure composed of:
    SQLDAID -- character string of length 8
    SQLDABC -- 31-bit binary integer
    SQLN    -- 15-bit binary integer
    SQLD    -- 15-bit binary integer
    SQLVAR  -- an array composed of:
        SQLTYPE -- 15-bit binary integer
        SQLLEN  -- 15-bit binary integer
            SQLPRCSN -- 1-byte (used for DECIMAL)
            SQLSCALE -- 1-byte (used for DECIMAL)
        SQLDATA -- 31-bit binary integer (pointer)
        SQLIND  -- 31-bit binary integer (pointer)
        SQLNAME -- varying-length character string
                of up to 30 characters
```

Figure 55. SQLDA Structure (in Pseudocode)

Note: The SQLLEN field can be divided into two subfields. The subfields are used only when working with DECIMAL values. Such usage is described in the following section.

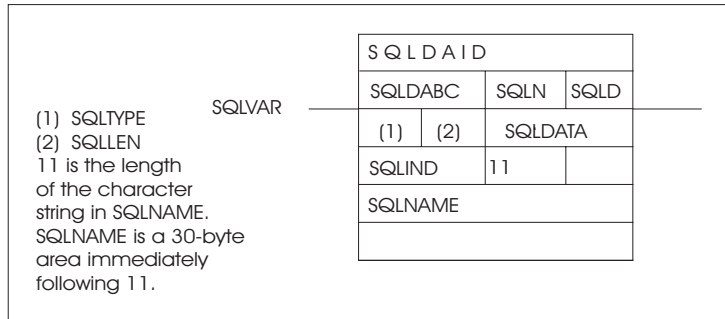
To include the declaration of the descriptor area in an assembler, C, or PL/I program, specify:

```
INCLUDE SQLDA
```

The INCLUDE SQLDA statement must not be placed in the SQL declare section. As with the SQLCA, you can code this structure directly instead of using the INCLUDE SQLDA statement. If you choose to declare the structure directly, you can specify any name for it. For example, you can call it SPACE1 or DAREA instead of SQLDA.

Processing a Run-Time Query Using the SQLDA

To process a run-time query, you must declare the SQLDA structure. Below is an illustration showing the SQLDA structure as a box; similar illustrations are used in following examples. Remember that SQLDA is a based structure (or, in assembler, a DSECT); no storage has actually been allocated yet.



The meanings of the various fields are described as they are used. A summary of the meanings of the fields of the SQLDA is presented later for quick reference.

If a *select-statement* is assigned to the variable QSTRING, it can be read in from SYSIPT or assigned within the program itself. In this example, the following *select-statement* is read in from the terminal:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO = 221
```

This *select-statement* has no INTO clause. When it is read in, it is assigned to the host variable QSTRING, which is then preprocessed by the PREPARE statement:

```
READ QSTRING FROM TERMINAL
EXEC SQL PREPARE S1 FROM :QSTRING
```

Allocating Storage for the SQLDA Using the SQLVAR Array

Now you can allocate storage for the SQLDA. The techniques for acquiring storage are language dependent; refer to the appropriate compiler or assembler manual.

Note: The usage of the SQLDA depends on the USING clause option of the DESCRIBE statement (discussed later in this chapter). In this section, it is assumed that the NAMES option of the USING clause has been specified. The amount of storage you need to allocate depends upon how many elements you want to have in the SQLVAR array. Each *select_list* item must have a corresponding SQLVAR array element. Therefore, the number of *select_list* items determines how many SQLVAR array elements you should allocate. However, because *select-statements* are specified at run time, it is not possible to know how many *select_list* items there will be. Consequently, you must guess.

Suppose, in this example, that no more than three items are ever expected in the *select_list*. This means that the SQLVAR array should have a dimension of three, because each item in a *select_list* must have a corresponding entry in SQLVAR.

Initializing the SQLN Field of the SQLDA

Having allocated an SQLDA of what you hope will be adequate size, you must now initialize the SQLDA field called SQLN. SQLN is set to the number of SQLVAR array elements you have allocated (that is, SQLN is the dimension of the SQLVAR array). In this example, you must set SQLN to 3. Here's the pseudocode for what was done so far:

```
Allocate an SQLDA of size 3
SQLN = 3
```


Inserting Values in the SQLDA

Having allocated storage, you can now DESCRIBE the statement. (Make sure that SQLN is set before the DESCRIBE.)

```
DESCRIBE S1 INTO SQLDA
```

When the DESCRIBE is executed, the system places values in the SQLDA. These values provide information about the *select_list*.

Figure 56 shows the contents of the SQLDA after the DESCRIBE is executed for the example *select-statement*. The third SQLVAR element is not shown because it was not used.

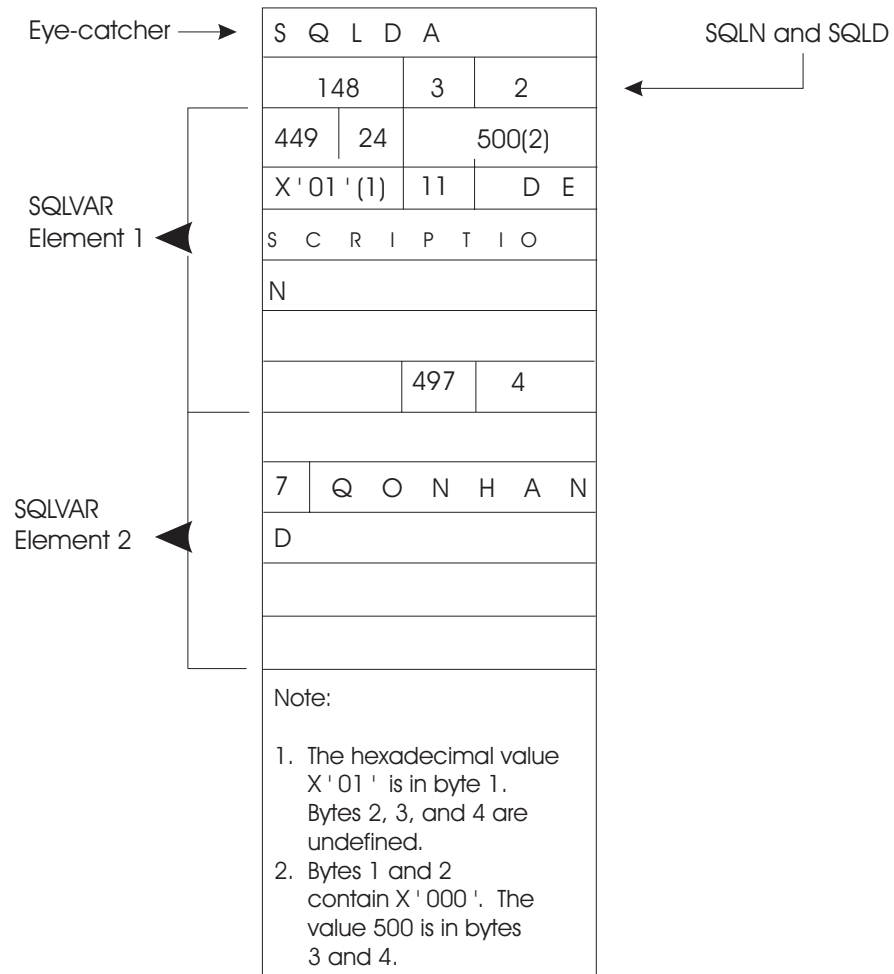


Figure 56. Contents of SQLDA after Executing the DESCRIBE

The SQLDAID and SQLDABC fields are initialized by the system when a DESCRIBE statement is executed (you can ignore these for now).

If you do not allocate a large enough SQLDA structure, SQLD will be set to the number of required SQLVAR elements after the DESCRIBE. Suppose, for example, that the *select-statement* contained four *select_list* expressions instead of two. The SQLDA was allocated with an SQLVAR dimension of three. The system cannot describe the entire *select_list* because there is not enough storage. In this case, SQLD is set to the actual number of *select_list* expressions; the rest of the structure is ignored. Thus, after a DESCRIBE it is a good practice to check SQLN.

If SQLN is less than SQLD, you need to allocate a larger SQLDA based on the value in SQLD:

```
EXEC SQL DESCRIBE S1 INTO SQLDA
IF (SQLN < SQLD)
    Allocate a larger SQLDA using the value of SQLD.
    Reset SQLN to the larger value.
EXEC SQL DESCRIBE S1 INTO SQLDA
END-IF
```

For the example *select-statement*, however, the SQLDA was of adequate size. SQLVAR has a dimension of three, and there are only two *select_list* expressions. SQLN remains set to 3, and SQLD is set to 2.

If you use DESCRIBE on a non-select-statement, SQLD is set to 0. If your program is designed to process both query and non-query statements, you can describe each statement (after it is prepared) to determine whether it is a query. This example routine is designed to process only query statements, so no test is provided.

Analyzing the Elements of SQLVAR

Your program must now analyze the elements of SQLVAR. Remember that each element describes a single *select_list* expression. Consider again the select-statement that is being processed:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO = 221
```

The first item in the *select_list* is DESCRIPTION. As illustrated in the beginning of this section, each SQLVAR element contains the fields SQLTYPE, SQLLEN, SQLDATA, SQLIND, and SQLNAME. The system returns a code in SQLTYPE that describes the data type of the expression and tells you whether nulls are applicable. For a detailed explanation on how to interpret the codes returned in SQLTYPE, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

For example, SQLTYPE is set to 449 in the first SQLVAR element. This indicates that DESCRIPTION is a VARCHAR column and that nulls are permitted in the column.

The system sets SQLLEN to the length of the column. For character strings, SQLLEN is set to the maximum number of bytes of the string. For graphic strings, SQLLEN is set to the maximum number of double-byte characters in the string. For decimal data, the precision and scale are returned in the first and second bytes, respectively. (Recall that the SQLLEN field has two sub-fields called SQLPRCSN and SQLSCALE for this purpose.) For other data types, SQLLEN is set as follows:

SMALLINT	-- SQLLEN = 2
INTEGER	-- SQLLEN = 4
Single precision float	-- SQLLEN = 4
Double precision float	-- SQLLEN = 8
DATE	-- SQLLEN = 10 or LOCAL
TIME	-- SQLLEN = 8 or LOCAL
TIMESTAMP	-- SQLLEN = 26

Note: For DATE, TIME, and TIMESTAMP, see “Using Datetime Data Types” on page 45.

Because the data type of DESCRIPTION is VARCHAR, SQLLEN is set equal to the maximum length of the character string. For DESCRIPTION, that length is 24. When the select-statement is later executed, a storage area large enough to hold a VARCHAR(24) string will be needed. In addition, because nulls are permitted in DESCRIPTION, a storage area for a null indicator variable would also be needed.

For character and graphic string columns, the system puts the CCSID attribute of the column in bytes 3 and 4 of the SQLDATA field. In Figure 56 on page 167, DESCRIPTION is a character column; therefore, the CCSID of DESCRIPTION is stored in the SQLDATA field of element 1. The example shows a CCSID of 500, which means that the data stored in the column is stored in CCSID 500 format.

For character string columns, the database manager stores an indicator in byte 1 of the SQLIND field. The indicator is set according to the subtype associated with the column. In Figure 56 on page 167, the indicator for DESCRIPTION is set to X'01', which means that DESCRIPTION has a subtype of SBCS. Columns with a subtype of SBCS can contain single-byte character set characters only.

The last field in an SQLVAR element is a varying-length character string called SQLNAME. The first two bytes of SQLNAME contain the length of the character data. The character data itself is usually the name of the field used in the *select_list* expression (DESCRIPTION in the above example). The exceptions to this are *select_list* items that are unnamed, such as functions (for example, SUM(SALARY)) and expressions (A+B-C). These exceptions are described in greater detail under “Summarizing the Fields of the SQLDA” on page 179.

The second SQLVAR element in the above example contains the information for the QONHAND *select_list* item. The 497 code in SQLTYPE indicates that QONHAND is an INTEGER column that permits nulls. For an INTEGER data type, SQLLEN is set to 4. SQLNAME contains the character string QONHAND, and has the length byte set to 7.

Allocating Storage for the Result of the Select-Statement

After analyzing the result of the DESCRIBE, you can allocate storage for variables that will contain the result of the select-statement. For DESCRIPTION, a varying character field of length 24 must be allocated; for QONHAND, a binary integer of 31 bits (plus sign) must be allocated. Both QONHAND and DESCRIPTION permit nulls, so you must allocate two additional halfwords to function as indicator variables.

After the storage is allocated, you must change the SQLDA. For each element of the SQLVAR array, do the following:

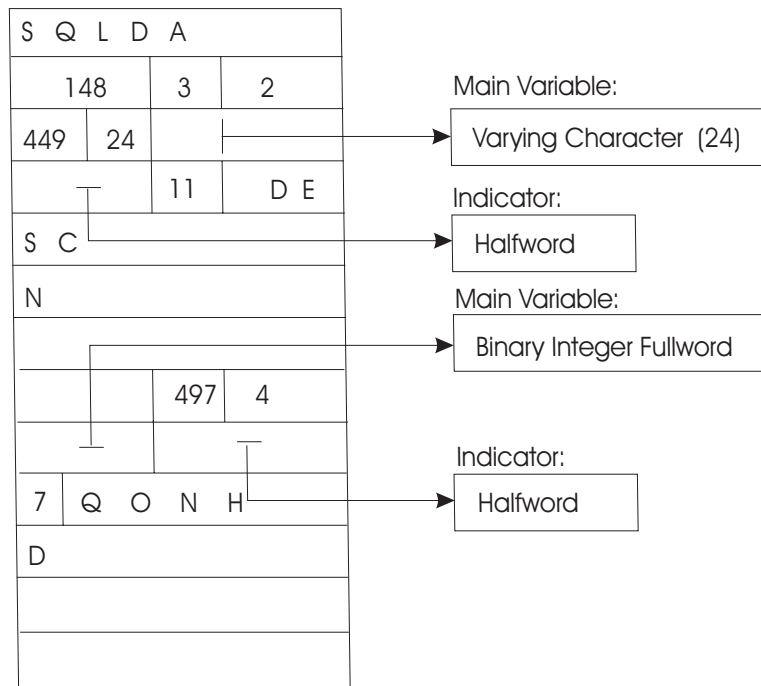
- Set SQLDATA to the address of the area in which the results will be placed.
- Set SQLIND to the address of the area in which the indicator information will be placed.
- If the data type of the area in which the results will be stored is character or graphic *and* you want to override the CCSID of the data area with, for example, the CCSID of the column, you must do the following:
 - Change the 6th position of the SQLDAID field to '+'. For example, set the SQLDAID field to 'SQLDA+ '.
 - Set the length of the SQLNAME field to 8. (The length here is not the SQLLEN field, but is the length associated with the SQLNAME field. For

example, in the SQLDA defined for C, the field referred to is sqlname.length.)

- Initialize bytes 1 and 2 of the SQLNAME field to 0
- Put the CCSID override value in bytes 3 and 4 of the SQLNAME field.

Note: When no override is present, the CCSID of the data area defaults to the application requester's default.

In the following example, the SQLDA is updated to contain the appropriate addresses. Because a CCSID override is not required, the SQLNAME field is not modified. Here is what the structure now looks like:



This is the pseudocode for what was done so far:

```

EXEC SQL INCLUDE SQLDA
.
.
READ QSTRING FROM TERMINAL
EXEC SQL PREPARE S1 FROM :QSTRING
Allocate an SQLDA of size 3.
SQLN = 3
EXEC SQL DESCRIBE S1 INTO SQLDA
IF (SQLN < SQLD)
    Allocate a larger SQLDA using the value of SQLD.
    Reset SQLN to the larger value.
    EXEC SQL DESCRIBE S1 INTO SQLDA
END-IF
Analyze the results of the DESCRIBE.
Allocate storage to hold select_list results.
Set SQLDATA and SQLIND for each select_list item.

```

Retrieving the Query Result

Now comes the easy part: retrieving the query result. Dynamically defined queries, as noted earlier, must not have an INTO clause. Thus, all dynamically defined queries must use a cursor. Special forms of the DECLARE, OPEN, and FETCH statements are used for dynamically defined queries.

The DECLARE CURSOR statement for the example query is as follows:

```
DECLARE C1 CURSOR FOR S1
```

The only difference is that the name of the prepared select-statement (S1) is used instead of the select-statement.

The actual retrieval of result rows is as follows:

```
EXEC SQL OPEN C1
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
DO WHILE (SQLCODE = 0)
    DISPLAY (results pointed to by SQLDATA and SQLIND
            for all pertinent SQLVAR elements)
    EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
END-DO
DISPLAY ('END OF LIST')
EXEC SQL CLOSE C1
```

The cursor is opened, and the result table is evaluated. (Note that there are no input host variables needed for the example query. Methods of providing input host variables are discussed later.) The query result rows are then returned using a FETCH statement (which does not have output host variables in this example). This statement returns results into the data areas referenced in the descriptor called SQLDA. The same SQLDA that was set up by DESCRIBE is now being used for the *output* of the select-statement.

The next section describes a more general routine in which you can process queries that have parameters in the WHERE clause. You should not read that section until you have coded some of the simpler dynamic queries discussed thus far.

Executing a Parameterized SELECT Statement

In the example above, the query that was dynamically executed had no parameters (input host variables) in the WHERE clause:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO = 221
```

Suppose you wanted to execute the same query a number of times using different values for PARTNO. A parameterized SQL statement is needed:

```
SELECT DESCRIPTION, QONHAND FROM INVENTORY WHERE PARTNO = ?
```

Generating an Additional SELECT Statement

In previous parameterized SQL statements, the number of parameters and their data types had to be known. What if they are unknown? The DESCRIBE statement, at first glance, is not feasible because it describes only *select_lists*. With some additional programming, however, you *can* use the DESCRIBE statement to obtain information about the parameter markers (?). Specifically, the code must scan the FROM and WHERE clauses to determine the table and column with which the parameter marker (?) is associated. The code can then construct a

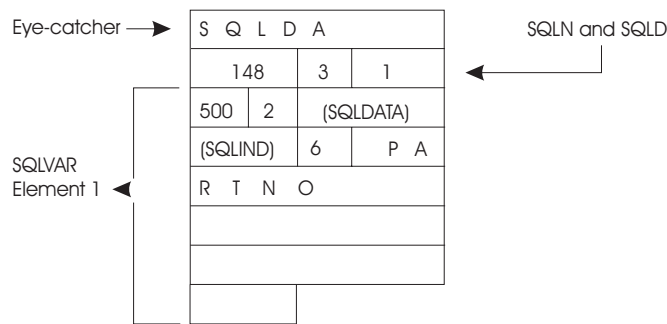
select-statement using those column names in the *select_list*. For the parameterized statement above, the following query can be generated:

```
SELECT PARTNO FROM INVENTORY
```

The query (assigned to WSTRING below) can then be preprocessed and described:

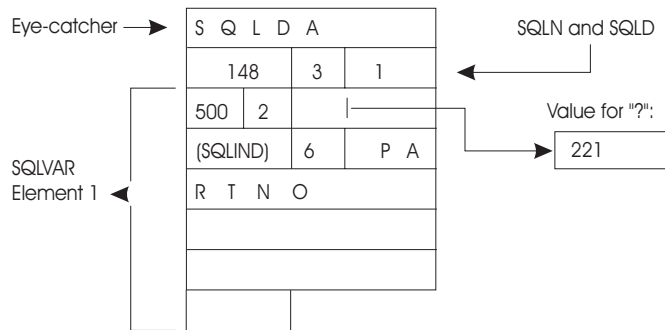
```
Allocate an SQLDA of size 3.
SQLN = 3
EXEC SQL PREPARE S2 FROM :WSTRING
EXEC SQL DESCRIBE S2 INTO SQLDA
```

Here is what the SQLDA looks like after the fabricated *select-statement* is described. Only the first element of SQLVAR is shown because the others are not used:



An analysis of the SQLDA shows that there is only one parameter marker (?), and that parameter is associated with PARTNO. The SQLTYPE value (500) indicates that PARTNO contains integer halfwords. Thus, you need to allocate a binary integer halfword for the parameter marker (?) variable. SQLDATA must then be set to point to this area.

Previously, the SQLDA was used in a FETCH statement, and query results were returned into the storage areas pointed to by SQLDATA and SQLIND. In other words, the SQLDA was used for *output*. Now, the SQLDA is going to be used to provide *input* values for the WHERE clause by an OPEN statement. When the SQLDA is being used for input, you must assign values to the dynamically allocated storage areas pointed to by SQLDATA. If the SQLTYPE value returned by DESCRIBE indicates that the field permits nulls, you must either supply an indicator variable pointed to by SQLIND, or reset SQLTYPE to indicate that nulls are *not* permitted. If indicator variables are not required, you should reset SQLTYPE. For example, if the SQLTYPE returned by DESCRIBE is 501, you should set it to 500 before using the SQLDA to provide input. After the storage for the parameter markers is allocated, you should read in values and assign them to those areas. Here is the completed SQLDA (assuming 221 is read in for the parameter marker (?)):



After an SQLDA is set up in this fashion, it can be referred to in an OPEN statement that contains a USING clause. For example, a previously declared cursor called C1 is opened using SQLDA:

```
OPEN C1 USING DESCRIPTOR SQLDA
```

Because SQLDA currently has 221 in the field pointed to by SQLDATA, C1 is evaluated using that value.

Figure 57 on page 174 shows the pseudocode for the complete example. Two SQLDA-like structures are used. One is called SQLDA, and is the usual structure; the other (declared directly) is called SQLDA1. The fields of SQLDA1 are suffixed with a "1"; for example, SQLDATA1 and SQLN1. An asterisk in position 1 of the pseudocode denotes a comment.

```

EXEC SQL INCLUDE SQLDA
Directly declare SQLDA1.
    .
    .
    .
* Read in a parameterized query.
*
  READ QSTRING FROM TERMINAL
*
* PREPARE and DESCRIBE the query; set up the output SQLDA.
*
  EXEC SQL PREPARE S1 FROM :QSTRING
  Allocate an SQLDA of size 3.
  SQLN = 3
  EXEC SQL DESCRIBE S1 INTO SQLDA
  IF (SQLN < SQLD)
    Allocate a larger SQLDA using the value of SQLD.
    Reset SQLN to the larger value.
    EXEC SQL DESCRIBE S1 INTO SQLDA
  END-IF
  Analyze the results of the DESCRIBE.
  Allocate storage to hold select list results.
  Set SQLDATA and SQLIND for each select_list item.
*
* Declare a cursor.
*
  EXEC SQL DECLARE C1 CURSOR FOR S1
*
* Fabricate a query so PREPARE and DESCRIBE can be used to
* set up the input SQLDA1.
*
  Scan the FROM clause and the WHERE clause of QSTRING for
  parameter markers (?) and generate an appropriate
  query in WSTRING.
  Allocate an SQLDA1 of size 1 (1 was obtained from the scan).
  SQLN1 = 1
  EXEC SQL PREPARE S2 FROM :WSTRING
  EXEC SQL DESCRIBE S2 INTO SQLDA1
  Analyze the results of the DESCRIBE.
  Reset SQLTYPE1 to reflect that there is no indicator variable.
  Allocate storage to hold the input values (the parameter marker (?)
  values).
  Set SQLDATA1 for each parameter marker (?) value.
*
* Read in input parameters and retrieve the query results using
* cursor C1. Note that the pseudocode reads in only one parameter
* marker (?). Your actual code must provide for the possibility
* that more than one parameter marker (?) might be provided.
*

```

Figure 57 (Part 1 of 2). Parameterized Query Statement


```

READ PARM FROM TERMINAL
DO WHILE (PARM ^= 0)
    Assign PARM to area pointed to by SQLDATA1.
    EXEC SQL OPEN C1 USING DESCRIPTOR SQLDA1
    EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
    DO WHILE (SQLCODE = 0)
        DISPLAY (results pointed to by SQLDATA and SQLIND)
        EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
    END-DO
    EXEC SQL CLOSE C1
    DISPLAY ('ENTER ANOTHER VALUE OR 0')
    READ PARM FROM TERMINAL
END-DO
DISPLAY ('END OF QUERY')

```

Figure 57 (Part 2 of 2). Parameterized Query Statement

Executing a Parameterized Non-Query Statement

“Executing Parameterized Statements” on page 162, introduces parameterized statements, however, it is necessary to know the number of parameter markers (?) and their data types before run time. The preceding section shows how you can analyze a parameterized query so that a *select-statement* can be generated and subsequently described. The same principle can be used for parameterized non-query statements.

Generating a SELECT Statement

For example, suppose this DELETE statement is read from the terminal and assigned to DSTRING:

```
DELETE FROM QUOTATIONS WHERE PARTNO = ? AND SUPPNO = ?
```

Suppose also that the number of parameter markers (?) and their corresponding data types are unknown before run time. The same routine that you coded to scan the FROM and WHERE clauses of select-statements can be used to scan the above DELETE statement. Then, a SELECT statement containing the relevant columns can be constructed:

```
SELECT PARTNO, SUPPNO FROM QUOTATIONS
```

This select-statement is then prepared and described as in the previous section. The setup of the SQLDA is also identical: once the SQLDA is analyzed, space to hold the parameter marker values is allocated, and these values are read in and assigned to these locations. The SQLDA will be used for *input* to the WHERE clause of the SQL statement; no indicator variables are allowed. Because the statement is a non-query statement, the SQLDA is pointed to in the EXECUTE statement. Figure 58 on page 176 illustrates the pseudocode for a parameterized non-query statement.

```

EXEC SQL INCLUDE SQLDA
.
.
.
READ DSTRING FROM TERMINAL
Scan the FROM clause and the WHERE clause of DSTRING for
parameter markers (?) and generate an appropriate query
in WSTRING.
Allocate an SQLDA of size 2 (2 was obtained from the scan).
SQLN = 2
EXEC SQL PREPARE S2 FROM :WSTRING
EXEC SQL DESCRIBE S2 INTO SQLDA
Analyze the results of the DESCRIBE.
Reset SQLTYPE to reflect that there is no indicator variable.
Allocate storage to hold the input values (the parameter marker (?)
values).
Set SQLDATA for each parameter marker (?) value.

EXEC SQL PREPARE S1 FROM :DSTRING
Read parameter marker (?) values from the terminal.
* A zero parameter value terminates the DO loop.
DO WHILE (parameters  $\neq$  0)
    Assign the values to the storage allocated for
    input variables.
    EXEC SQL EXECUTE S1 USING DESCRIPTOR SQLDA
    Prompt user for more values.
    Read parameter marker (?) values from the terminal.
END-DO
.
.
.

```

Figure 58. Parameterized Non-Query Statement

You may need a more complex scanning routine, depending on how many different non-query statements you wish to process. For example, the above routine would have to be modified if you wanted to process INSERT statements. In that case, you would have to scan for the table and column names.

Note: Indicator variables are permitted when you are providing input to the INSERT statement with EXECUTE.

Using an Alternative to a Scanning Routine

In the previous sections on parameterized statements (both query and non-query), you must rely on a scanning routine to generate a query. Once the query is generated, DESCRIBE obtains information about the columns and expressions associated with a parameter marker.

If you have not coded a scanning routine that generates a query, there is a simple alternative: have the user describe the parameter markers for you, and fill in the SQLDA yourself. There is no rule that says you must use a DESCRIBE to fill in the SQLDA. When using the SQLDA for input or output, it does not matter what fills it in, as long as the needed values are there.

When you use the SQLDA for input (which is always the case for parameter markers), not all fields have to be filled in. Specifically, SQLDAID and SQLDABC need not be filled in. Thus, if you choose this method, you will need to ask the user for the following:

1. How many parameter markers (?) are there?
2. What are the data types and lengths of these parameters?

In addition, if the routine is to handle both query and non-query statements, you may want to ask the user what category of statement it is. (Alternatively, you can write code to look for the SELECT keyword.)

The code that interrogates the user and sets up the SQLDA would take the place of the scanning routine and DESCRIBE in the previous sections:

With a Scanning Routine:

```

      .
      .
      .
READ DSTRING FROM TERMINAL
Scan the FROM and WHERE clauses of DSTRING for parameter markers (?)
and generate an appropriate query in WSTRING.
Allocate an SQLDA of size 2 (2 was obtained from the scan).
SQLN = 2
      EXEC SQL PREPARE S2 FROM :WSTRING
      EXEC SQL DESCRIBE S2 INTO SQLDA
Analyze the results of the DESCRIBE.
Reset SQLTYPE to reflect that there is no indicator variable.
Allocate storage to hold the input values
      (the parameter marker (?) values).
Set SQLDATA for each parameter marker (?) value.
      .
      .
      .

```

Without a Scanning Routine:

```

      .
      .
      .
READ DSTRING FROM TERMINAL
Interrogate user for number of parameter markers (?).
Allocate an SQLDA of that size.
Set SQLN and SQLD to the number of parameter markers (?).
For each parameter marker (?):
      Interrogate user for data types, lengths, and
      indicators.
      Set SQLTYPE and SQLLEN.
      Allocate storage to hold the input values
      (the parameter marker (?) values).
      Set SQLDATA and SQLIND (if applicable) for each
      parameter marker (?).
      .
      .
      .

```

The statement can then be processed in the usual manner.

Ensuring Data Type Equivalence in a Dynamically Defined Query

In previous uses of the SQLDA for input or output, `SQLTYPE` *always* described the data type of the storage area pointed to by `SQLDATA`. In the following example, the type code 500 (originally obtained with a `DESCRIBE` of the select-statement) describes the data type of the main variable.

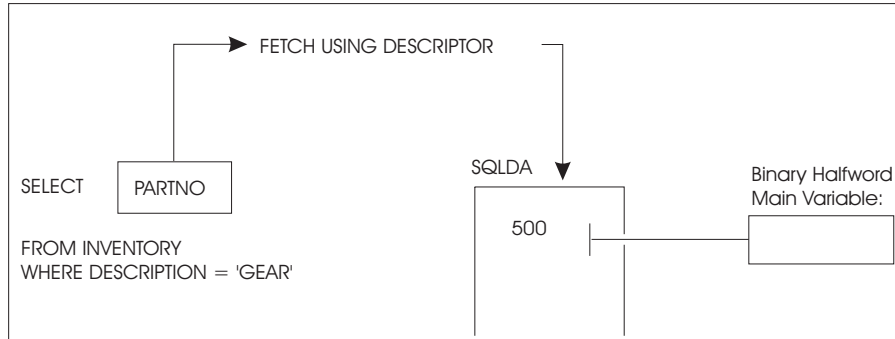


Figure 59. The `FETCH Using Descriptor`

In previous sections, the `select_list` item, the type code, and the data type of the storage area allocated for holding query results are all equivalent. That is, in the above example, `PARTNO` is a `SMALLINT` column (with no nulls permitted), 500 is the type code meaning `SMALLINT NOT NULL`, and the area allocated is a binary integer halfword. To force a data conversion, you must allocate a storage area having a different data type and then change `SQLTYPE` in the `SQLDA`. Suppose that you wanted to select the `SMALLINT` part numbers into an integer area. Here is the sequence of instructions needed:

```
EXEC SQL PREPARE S1 FROM :STRING
EXEC SQL DESCRIBE S1 INTO SQLDA
Allocate a binary integer fullword of storage.
Set SQLDATA to point to it.
SQLTYPE = 496
```

When the `FETCH` is executed, `SMALLINT` is converted to `INTEGER`. Similarly, you could have converted the retrieved `PARTNO` values to `FLOAT` merely by setting `SQLTYPE` to 480 and by allocating a floating-point word of storage.

This conversion can be done when the `SQLDA` is used for *input* also. Consider the normal case:

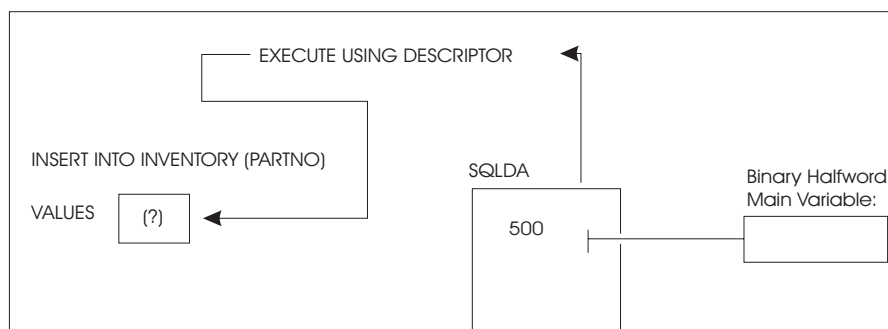


Figure 60. The `EXECUTE Using Descriptor`

As before, PARTNO is SMALLINT. The main variable is also allocated as SMALLINT (binary integer halfword), and the SQLTYPE that describes the main variable represents a SMALLINT. To perform data conversion on input, you need to change only the SQLTYPE and the type of storage allocated to hold the input values. This is done exactly as in the previous example. To insert a floating-point variable into the SMALLINT PARTNO column, for example, these steps are needed:

```
EXEC SQL PREPARE S1 FROM :STRING
EXEC SQL PREPARE S2 FROM 'SELECT PARTNO FROM INVENTORY'
EXEC SQL DESCRIBE S2 INTO SQLDA
Allocate an 8-byte floating-point area.
Set SQLDATA to point to it.
Assign a floating-point number to the area.
SQLTYPE = 480
EXEC SQL EXECUTE S1 USING DESCRIPTOR SQLDA
```

All dynamic data conversion is done according to the rules summarized under “Converting Data” on page 48.

If you change the SQLTYPE code and then allocate a storage area of an incorrect type, the system treats the storage area as though it were of the type indicated by SQLTYPE. For example, suppose SQLTYPE indicates that the storage area pointed to by SQLDATA is an INTEGER, but that the actual area allocated is a binary integer halfword (SMALLINT). The field is treated as though it is an INTEGER, not a SMALLINT. This type of error may yield confusing results.

When a datetime data code is used in an SQLDA on a FETCH, the system assumes that the variable declared to hold the result is fixed-length character.

Summarizing the Fields of the SQLDA

This section summarizes the SQLDA structure and related information.

As you have learned in the previous sections, the SQLDA can be used in any number of ways. In general, the fields within the SQLDA must be initialized either by using a DESCRIBE statement or by user code. Once they are initialized, the SQLDA can be used for *input* (in EXECUTE, OPEN, and PUT) or for *output* (in FETCH).

summarizes the sequence of events needed to initialize the SQLDA for use in processing dynamically defined statements. In any case, you must always initialize SQLN before the DESCRIBE.

If you do not use a DESCRIBE to set up the SQLDA, you need only fill in those fields that are actually used by the OPEN, FETCH, PUT, or EXECUTE statements.

For applications that override the defaults for subtypes and CCSIDs, the SQLDA provides output information on subtypes and CCSIDs. The *DB2 Server for VSE & VM SQL Reference* manual contains a description of the structure of the SQLDA, and an explanation of each field within the SQLDA. The following are some additional guidelines for using the SQLN and SQLD fields.

Sequence of Events →				
SQLDA Fields:	First, DESCRIBE initializes:	Then you must initialize:	Next, if you intend to use the SQLDA for input (EXECUTE or OPEN), you must place values in the locations pointed to by SQLDATA and SQLIND. When the SQLDA is used for output (FETCH), the system places values in those areas.	EXECUTE, OPEN, PUT and FETCH use:
SQLDAID(3)	X			
SQLDABC	X			
SQLN(1)				
SQLD	X			X
SQLVAR				
SQLTYPE	X			X
SQLLEN	X			X
SQLDATA	X	X		X
SQLIND(2)	X	X		
SQLNAME(3)	X	X		X

Notes:

1. You must set SQLN before the DESCRIBE.
2. Only provide indicator variables if they are allowed. In dynamic SQL, indicator variables should be used for output. They can be used for input in an INSERT or UPDATE, but not in predicates.
3. Only update the SQLDAID and SQLNAME fields if a CCSID override is required. The database manager extracts the CCSID from the 3rd and 4th byte of the SQLNAME field only when the following are true:
 - * The data type of the user data area is character or graphic
 - * The 6th byte of the SQLDAID field has been set to '+'.
For example, the SQLDAID field is 'SQLDA+ '.
 - * The length of the SQLNAME field is 8
 - * The first two bytes of data in the SQLNAME field are X '0000 '.

Figure 61. SQLDA Initialization

Using the SQLN Field

Always set this value when the structure is allocated. When the USING clause of the DESCRIBE statement is set to NAMES, LABELS, or ANY, specify the maximum number of expected *select_list* items. When you set the USING clause option to BOTH, specify twice the number of expected *select_list* items.

Using the SQLD Field in the SQLDA

If the statement being described is not a *select-statement*, the database manager returns a zero in SQLD. If the statement is a *select-statement*, SQLD is set to indicate the number of SQLVAR elements. This number is either the number of *select_list* elements (when the USING clause of the DESCRIBE statement is set to NAMES, LABELS, or ANY), or twice the number of *select_list* elements (if the USING clause is set to BOTH).

In the second case (USING clause set to BOTH), your program should reset SQLD to half its value before issuing a subsequent FETCH or PUT. This is because only the first N/2 elements contain information; the rest contains label information only.

If (after a DESCRIBE) SQLD is greater than SQLN, the SQLVAR array is not large enough to contain descriptions for all the *select_list* items. In this case, you must allocate a larger SQLDA based on the value of SQLD. The value in SQLN is not changed.

If you set the value of SQLD yourself, and you set it to less than SQLN, the excess elements of the SQLVAR array are ignored.

Using the PREPARE Statement

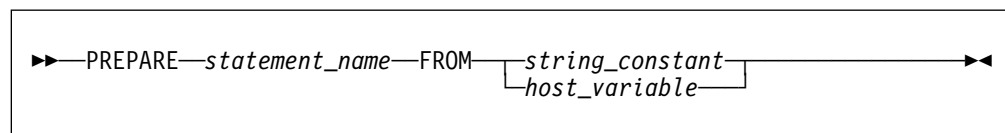


Figure 62. Format of the PREPARE statement

Although a statement to be “prepared” cannot contain any host variables, it can contain parameters to be filled in when the statement is executed. These parameters are denoted by parameter markers (?). You can specify parameters only in places where a data value could be used. (A parameter cannot represent the name of a table or a column.) The pseudocode example below prepares an INSERT statement that has three parameters:

```
QSTRING='INSERT INTO DEPARTMENT(DEPTNO,DEPTNAME,ADMRDEPT) VALUES (?,?,?)'
```

```
PREPARE S1 FROM :QSTRING
```

Each time S1 is executed, values must be supplied for the three parameters that were specified with question marks.

If your program constructs dynamic SQL statements by manipulating quoted strings, remember that SQL uses two single quotation marks to represent a quotation mark inside a quoted string. The following example illustrates this rule:

```
PREPARE S1 FROM 'INSERT INTO DEPARTMENT(DEPTNO,DEPTNAME,ADMRDEPT)
VALUES ('A00','SPIFFY COMPUTER SERVICE DIV.','A00')'
```

In this example, the text beginning with INSERT and ending with A00') is a constant string. Each pair of quotation marks is collapsed to a single quotation mark.

In COBOL, a constant string-spec is treated as a COBOL character string and is affected by the Quote/APOST option. This option determines the character string delimiters. If you use the same character (" or ') in the constant string-spec as the one established by Quote/APOST option for the outer string delimiters, unexpected string termination can result.

It is best to avoid using a constant string-spec whenever it may contain quotation marks. Instead, you should build the SQL statement as a host variable string-spec, using the known host language rules for character strings. For SQL statements that contain graphic constants, be aware that some DBCS characters may contain the encodings for EBCDIC quote. This could cause unintentional termination of host language strings that contain DBCS characters.

A parameter marker (?) can appear in an SQL statement to be “prepared” in any place that a host variable may appear, with the following exceptions:

- A parameter marker cannot be used in a *select_list* or a FROM-clause (but it may be used in the WHERE clause of a SELECT statement).

The following examples are invalid:

```
SELECT ? FROM EMPLOYEE
SELECT EMPNO FROM ?
```

The following example is valid:

```
SELECT * FROM EMPLOYEE WHERE EMPNO = ?
```

- At least one of the operands of the arithmetic and comparison operators, or of the BETWEEN and IN predicates, must not be a parameter marker.

The following examples are invalid:

```
SELECT * FROM EMPLOYEE WHERE SALARY > ? + ?
SELECT * FROM EMPLOYEE WHERE ? = ?
SELECT * FROM EMPLOYEE WHERE ? IN (?,?)
```

The following examples are valid:

```
SELECT * FROM EMPLOYEE WHERE SALARY > 20000 + ?
SELECT * FROM EMPLOYEE WHERE SALARY = ?
SELECT * FROM EMPLOYEE WHERE ? IN (?,?,20000)
```

- A parameter marker cannot be the sole argument of a scalar function. It, however, can be used in an arithmetic expression as long as the other parameter is a number.

The following example is invalid:

```
SELECT * FROM EMPLOYEE WHERE HIREDATE > DATE(?)
```

The following example is valid:

```
SELECT * FROM EMPLOYEE WHERE HIREDATE > DATE(14+?)
```

- A parameter marker cannot be used as the sole operand in an arithmetic expression that involves a datetime value.

The following examples are invalid:

```
SELECT * FROM EMPLOYEE WHERE HIREDATE = START_DATE + ?
SELECT * FROM EMPLOYEE WHERE HIREDATE = 10000000. + ?
```

The following example is valid:

```
SELECT * FROM EMPLOYEE WHERE HIREDATE = HIREDATE + (1000000.+?)
```

SQL Functions Not Supported in Dynamic Statements

The following SQL functions are not supported in dynamic SQL:

- Syntax and semantic flagging of the dynamically executed statement
- SQL comments
- Negative indicator variables in predicates
- Optional choice for the FOR UPDATE OF clause in cursor query statements (NOFOR support).
- SQL CALL statement.

Chapter 7. Using Extended Dynamic Statements

Contents

Using Extended Dynamic Statements to Maintain Packages	184
Illustrating the Use of Extended Dynamic Statements	187
Developing a Query Application	187
Developing a Language Preprocessor	189
Grouping Extended Dynamic Statements in an LUW	193
Using COMMIT WORK and ROLLBACK WORK Statements	195
Considering Virtual Storage in a Logical Unit of Work	195
Mapping Extended Dynamic Statements to Static and Dynamic Statements	195
SQL Functions Not Supported in Extended Dynamic Statements	197

Using Extended Dynamic Statements to Maintain Packages

Extended dynamic statements support the direct creation and maintenance of packages for DB2 Server for VSE data. Extended dynamic statements can only be used with assembler language. manual).

Note: This topic is more advanced than previous sections and the techniques discussed here are not relevant to all application programs.

Before reading this chapter, you should be familiar with how to use packages as described in , “Preprocessing the Program” on page 108, and dynamically defined statements, as described in Chapter 6, “Using Dynamic Statements” on page 159. Extended dynamic statements provide a function similar to that provided by the DB2 Server for VSE preprocessors, but may be particularly useful where:

- The current preprocessors do not support the language of the application or support program.
- SQL statements are conceived and built dynamically, but are executed repetitively (in a different logical unit of work). In this case it is more efficient to avoid having to repeat the preprocessing of statements each time they are executed, as would be required for normal dynamic statements.
- You want to build and maintain an application package of SQL statements to be shared by a group of users.
- The utilization of program storage is critical and there are a significant number of predefined “transactions” involving DB2 Server for VSE data.

Individual SQL statements can be added or deleted without affecting or repeating the preprocessing of other SQL statements in the package.

The following extended dynamic statements are supported. (They are described in detail in the *DB2 Server for VSE & VM SQL Reference* manual.)

- CREATE PACKAGE—build an empty package
- PREPARE—add a statement to a package
- DESCRIBE—obtain information about columns in the *select_list* of a prepared *select-statement*
- EXECUTE—execute a statement in a package
- DECLARE CURSOR—in connection with OPEN, FETCH, PUT, and CLOSE, execute a SELECT or an INSERT statement in a package
- OPEN (cursor)
- FETCH (cursor)
- PUT (cursor)
- CLOSE (cursor)
- DROP STATEMENT—delete a statement from a package.

Except for CREATE PACKAGE and DROP STATEMENT, the names of these statements are the same as the corresponding “normal” dynamic statements discussed in Chapter 6, “Using Dynamic Statements” on page 159, but their format and meaning are somewhat different. For example, the statement-id, package-id, and cursor-name fields are all specified by host variables.

Unlike dynamic statements which are related through a specific statement name, extended dynamic statements are related through the symbolic host variables used for the statement-id and package-id. This relationship is shown in Figure 63. Because the statement-id and package-id are host variables, actual values can be substituted when the program is executed. STMTID is returned by an extended PREPARE statement, and is used as input by the subsequent extended EXECUTE (or DECLARE CURSOR) statement.

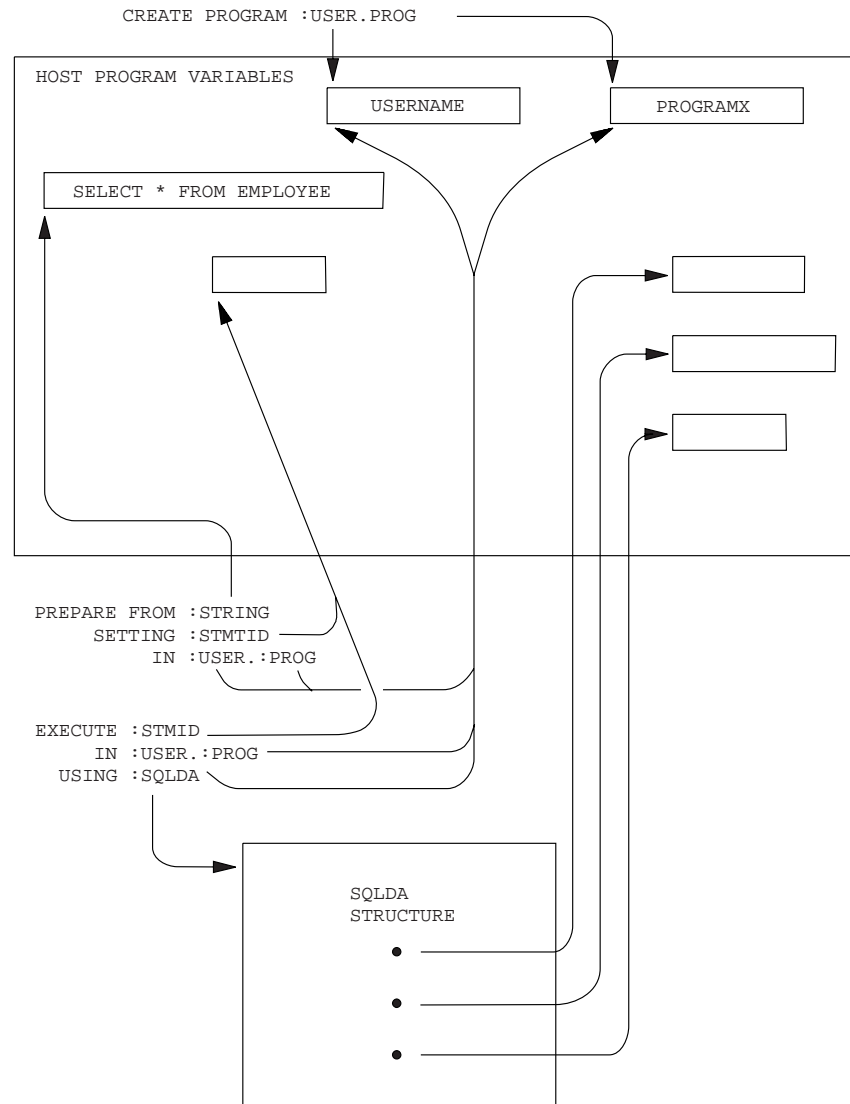


Figure 63. Relationship between Extended Dynamic Statements Expressed Using Host Program Variables

The differences between dynamic and extended dynamic statements are illustrated in Figure 64. As shown in this figure, the normal dynamic statements are intended primarily for supporting an interactive environment. As such, the PREPARE and EXECUTE commands must be used within the same logical unit of work. In contrast, extended dynamic statements are generally used in a compile environment where the EXECUTE (or DECLARE CURSOR) may be in a logical unit of work that is different from the one where the SQL statement was prepared. This makes it possible to PREPARE statements at different times. In DB2 Server for VSE terms, they can be prepared in one logical unit of work (stored in a

package), and called out for execution from another logical unit of work (from the same or a different program). This is made possible by passing the program and statement identifiers between the preparation environment and the execution environment.

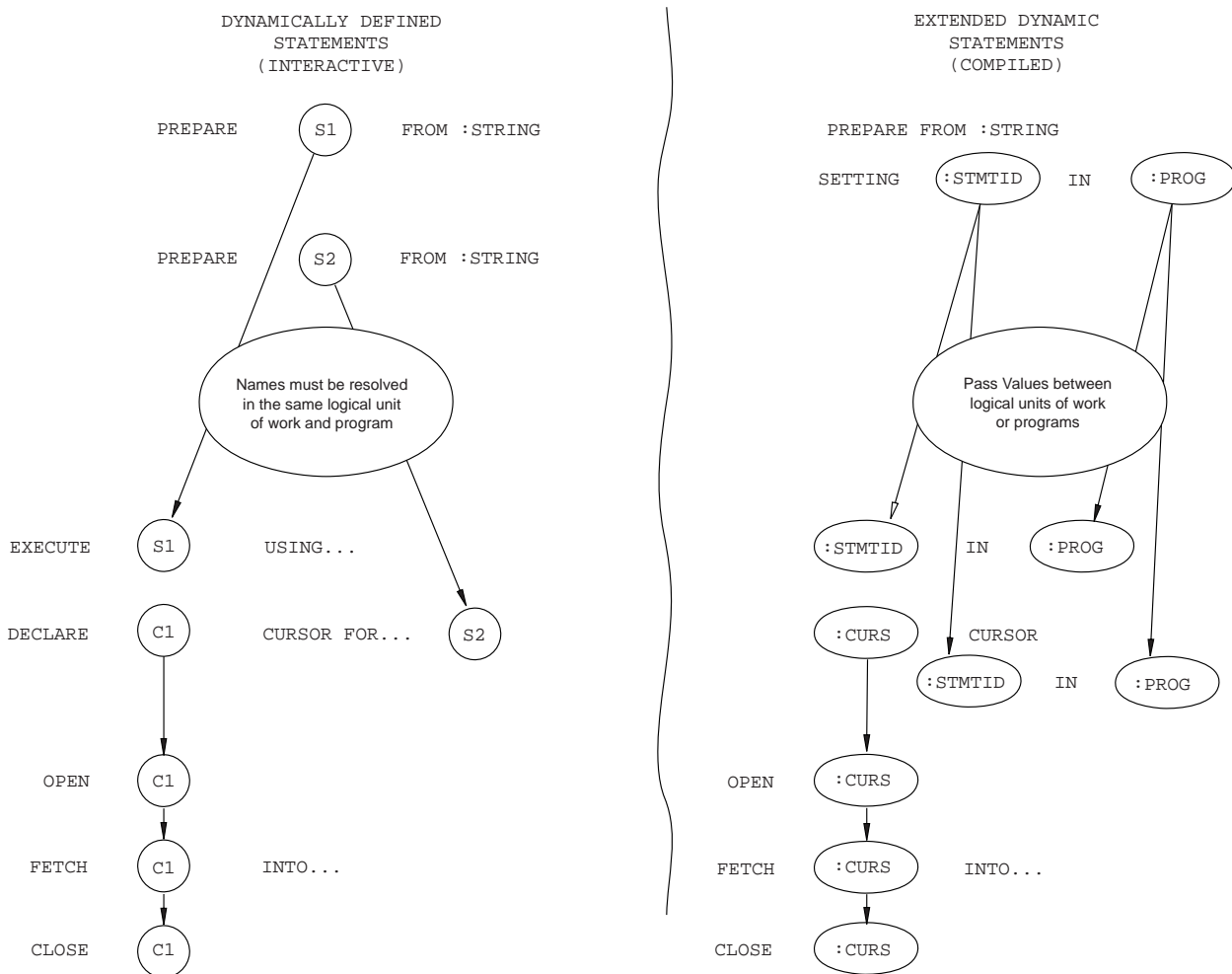


Figure 64. Comparing Dynamic to Extended Dynamic Statements

CREATE PACKAGE and DROP STATEMENT have no counterparts in the normal dynamic statement set.

CREATE PACKAGE creates an empty package and is normally followed by extended PREPARE statements to add statements to the package. If the CREATE PACKAGE has a MODIFY option, the package may even be changed in another logical unit of work. The change may take the form of additional extended PREPAREs (adding statements to those already there), or DROP STATEMENTS (deleting statements previously prepared). If a program is created with the NOMODIFY option, it cannot be changed without completely replacing it. You do this by using CREATE PACKAGE with the REPLACE option and specifying the same package-id.

The DROP PACKAGE statement is not listed as an extended dynamic statement, because it has general applicability for all packages, not just those that are built with extended dynamic statements. (See the *DB2 Server for VSE & VM SQL Reference* manual for more information on the DROP PACKAGE statement.) Like

the extended dynamic statements, DROP PACKAGE permits the package name to be specified as a host program variable.

Like all SQL statements, extended dynamic statements require preprocessing, but are only supported by the assembler preprocessor. Once they are preprocessed (and the containing program is compiled), the program holding them may itself be used to process SQL statements and create packages. That is, it may prepare SQL statements for repetitive execution. A package created in SQLDS protocol that uses extended dynamic statements is not supported in DRDA protocol, nor is a package created in DRDA protocol that uses extended dynamic statements supported in SQLDS protocol. The nonmodifiable environment, when using extended dynamic statements, is supported with the following restrictions:

- The Positioned UPDATE or Positioned DELETE statements are not supported.
- If you use the basic format of the extended PREPARE statement to prepare a statement that contains parameter markers, you must include the USING DESCRIPTOR clause to identify an input SQLDA structure.
- The prepare single row format of the extended PREPARE statement is not supported.
- The NODESCRIBE option of the CREATE PACKAGE statement is not supported.
- The temporary extended prepare format of the extended PREPARE statement is not supported.
- The using output descriptor clause in the extended EXECUTE statement is not supported.
- The USER parameter in the ISOLATION option of the CREATE PACKAGE statement is not supported.
- The LOCAL parameter in the DATE or TIME option of the CREATE PACKAGE statement is not supported.

Illustrating the Use of Extended Dynamic Statements

Developing a Query Application

Consider the following example. A support group needs to develop a program that dynamically accepts SQL statements for execution and does not know what SQL statements will be processed. This is a typical application for normal dynamic SQL statements. But since there is also a requirement for repetitively executing the preprocessed statements at a later time (stored SQL application) without having to repeat the PREPARE, it is an application for extended dynamic statements.

A program that handles preparation of end user SQL statements can also execute these statements. This is essentially a query language program (but it supports more than just select-statements). The program may also support deleting statements from and adding them to existing packages. (See the beginning of this chapter for a list of extended dynamic statements for doing this, as well as statements to control execution.)

The program may use CREATE PACKAGE and extended PREPARE to build a package and prepare the end-user SQL statements. However, you must first

preprocess the program itself, by running it through the assembler preprocessor and the assembler. See Figure 65 (the application program is referred to as a “Support Program”).

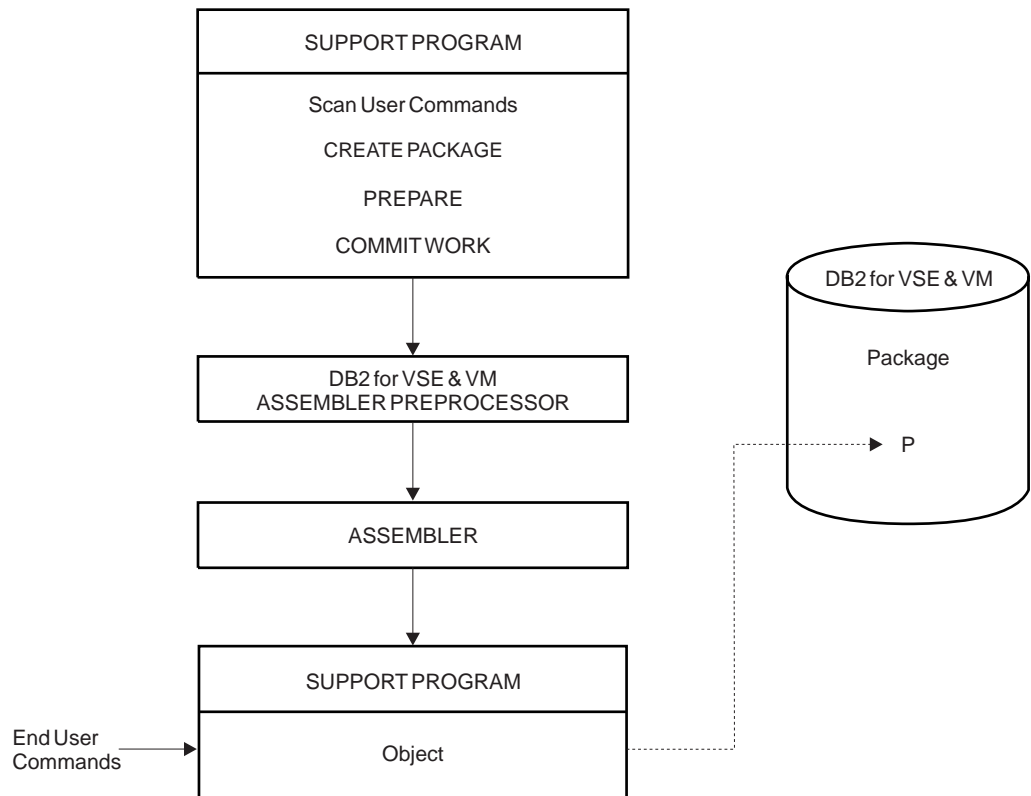


Figure 65. An Example of an Interpretive Support Program for Building and Executing SQL Statements in a Package

The resulting support program can accept end-user SQL statements, and create packages in the database to hold them. For example, there can be a separate package to hold the SQL statements for each end-user. A more advanced support program may even accept end-user commands that are at a higher level than that supported by the system, and then translate them to SQL statements before preparing them.

The package P is built by the support program (by CREATE PACKAGE) for the particular SQL statements. If the support program allows both adding SQL statements to and dropping them from P, then the support program must utilize and be preprocessed with a DROP STATEMENT as well as the PREPARE. Of course, there are a few other ordinary SQL statements that may be appropriate for the support program: WHENEVER, COMMIT/ROLLBACK, and so on to make it complete.

So far, this example has not addressed execution of the end-user SQL statements. We have already listed the extended dynamic statements that support execution (extended EXECUTE, DECLARE CURSOR, and so on). The support program would ordinarily support end-user commands to retrieve data and update data (using either direct SQL statements or higher level commands that require conversion). This addition does not alter the concept shown in Figure 65, except to add additional extended dynamic statements to the support program.

The DESCRIBE statement can be used in the same way as shown under normal dynamic statements.

Note that only one “copy” of each extended dynamic statement need be provided in the support program, because each of these statements is parameterized with host variables that can be dynamically changed for each use. For example, one DECLARE CURSOR statement may service all cursor retrievals, even if they are concurrently open, because each can be given a different cursor name by the host variable value for the cursor name, and a different statement identifier by the host variable value for the statement-id. This is important in cases where the use of program storage is critical and there are a significant number of predefined transactions.

Developing a Language Preprocessor

The previous example is structurally simple. It assumes that the support program remains in control as an interpreter through preparation, maintenance, and execution of the user's SQL statements.

For a typical language preprocessor program such as those provided with the DB2 Server for VSE product, however, this is not the case. If you write a support program for a new language preprocessor, you would probably separate the two parts, each with SQL statements:

1. One for preparation of end-user SQL statements and creation of a package.
2. Another for supporting the execution of the SQL statements that were prepared by the first part.

The SQL facilities required are similar to the previous example, except that no package maintenance functions are needed. The language preprocessor has the following characteristics:

- It is a batch program, rather than an interpreter.
- Because it requires extended dynamic statements, it is written in assembler language. (This was also true in the previous example.) Alternatively, at least part of it must be written in assembler language (the part that contains the extended dynamic statements) and the remaining part must be written in a language that is capable of calling an assembler module.
- Rather than accepting predefined commands from the end-user, the end-user's source language code is scanned for SQL statements, which must be identified by some defined convention (for example, EXEC SQL) for proper recognition.
- The support program must record information about host program variables in a control structure that is added to the end-user's source program and passed by a generated call to the execution-time part of the support program. This control structure builds SQLDA structures that are passed to or received from the system (refer to the extended EXECUTE, OPEN and FETCH statements).
- The execution part of the support program is link/loaded with the user's application program, where it is available to handle the execution-time functions.
- As each end-user SQL statement is prepared, the package-id and the statement-id (returned by the system along with the package-id) must be saved in a control structure (again generated into the end-user's source program) for use by the execution-time support program.

- For each SQL statement in the end-user's source program, a call must be generated to the execution-time support program, passing the control structure, containing the host variable, package-id, and statement-id information for the current SQL statement.
- The execution-time support program must build the SQLDA structures required, set values in host variables required by the execution-time extended dynamic statements, and then execute these statements.

This process is illustrated in Figure 66, Figure 67, Figure 68, and Figure 69. The support program is the preprocessor for language X. It preprocesses the end-user program, modifying the source (adding control structures and generating calls to pass to the support program Part 2 at execution-time). Once the modified end-user source has been compiled by the language X compiler, it is combined in one load module with the object code for the support program Part 2, which provides the DB2 Server for VSE support for execution-time functions (DECLARE CURSOR, and so on).

Figure 66 shows the preprocessing and assembly steps for the two parts of the support program.

Figure 67 on page 191 shows how the two resulting object modules of the support program process end-user SQL statements.

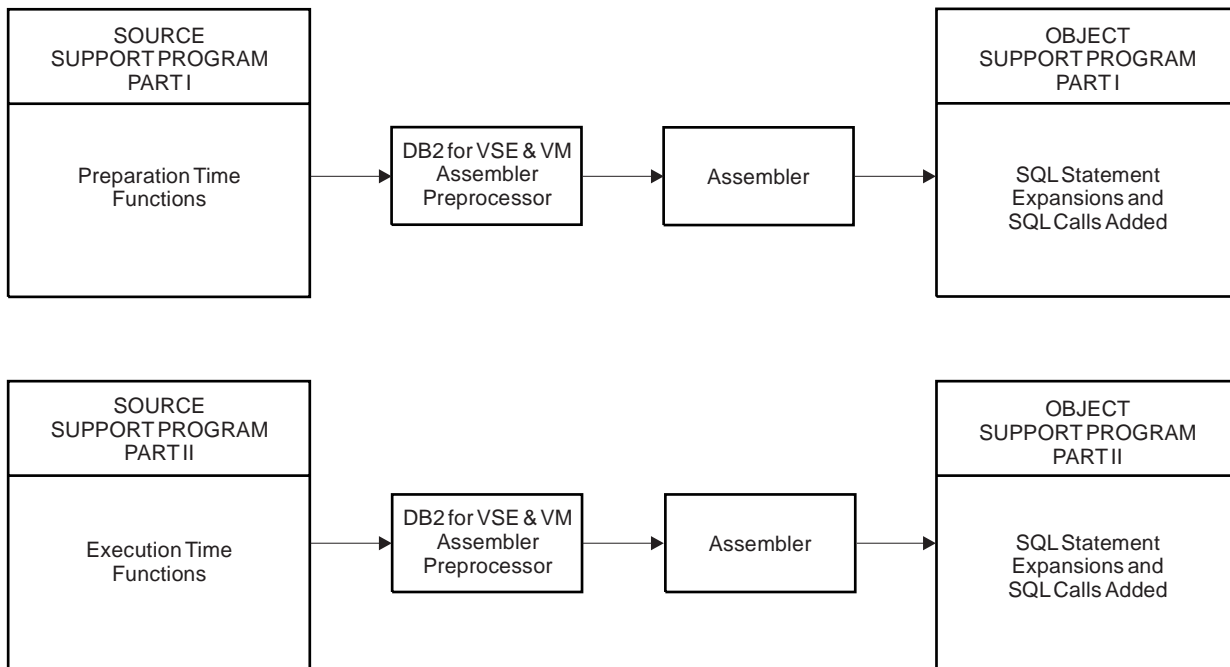


Figure 66. Preprocessing and Assembling of a Two-Part Support Program

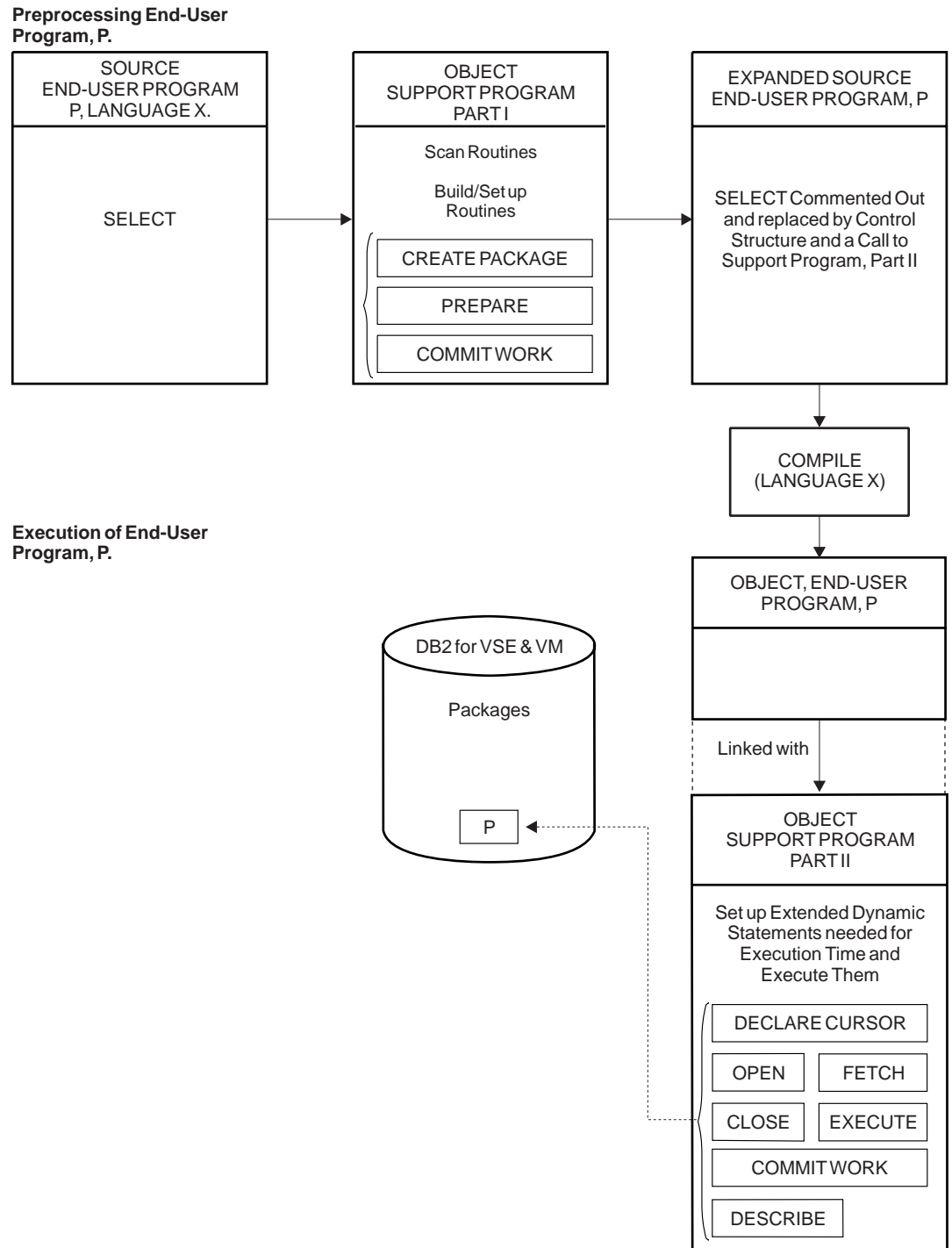


Figure 67. Preprocessing and Executing an End-User Program by a Two-Part Support Program

Part 1 scans the end-user's source for SQL statements, uses CREATE PACKAGE to build an empty package, P, uses extended PREPARE statements to add SQL statements to P, and uses a COMMIT statement to finalize P. It also adds calls and control structures, required by Part 2 of the support program, to the user's source program and comments out the original SQL statement.

Part 2 of the support program works with the package, P, executing the SQL statements scanned and prepared by Part 1, and using the control structures passed in the calls generated by Part 1. Part 2 must be link/loaded with any end-user module that is preprocessed by Part 1.

Figure 68 shows Part 1 of the support program in more detail, with pseudocode to illustrate a simple user program that includes a DECLARE...CURSOR FOR SELECT..., an OPEN of that cursor, and a FETCH for the same cursor. Control structures are shown in more detail, and some particular values for parameters are given. The value 26 returned from the PREPARE statement is only for purposes of illustration, representing a unique identifier returned by the system to identify the statement within the package P. A user ID may be necessary to identify the owner of the package, but it is omitted here for simplicity. Other statements, such as CLOSE (cursor) and COMMIT are not shown in order to simplify the illustration.

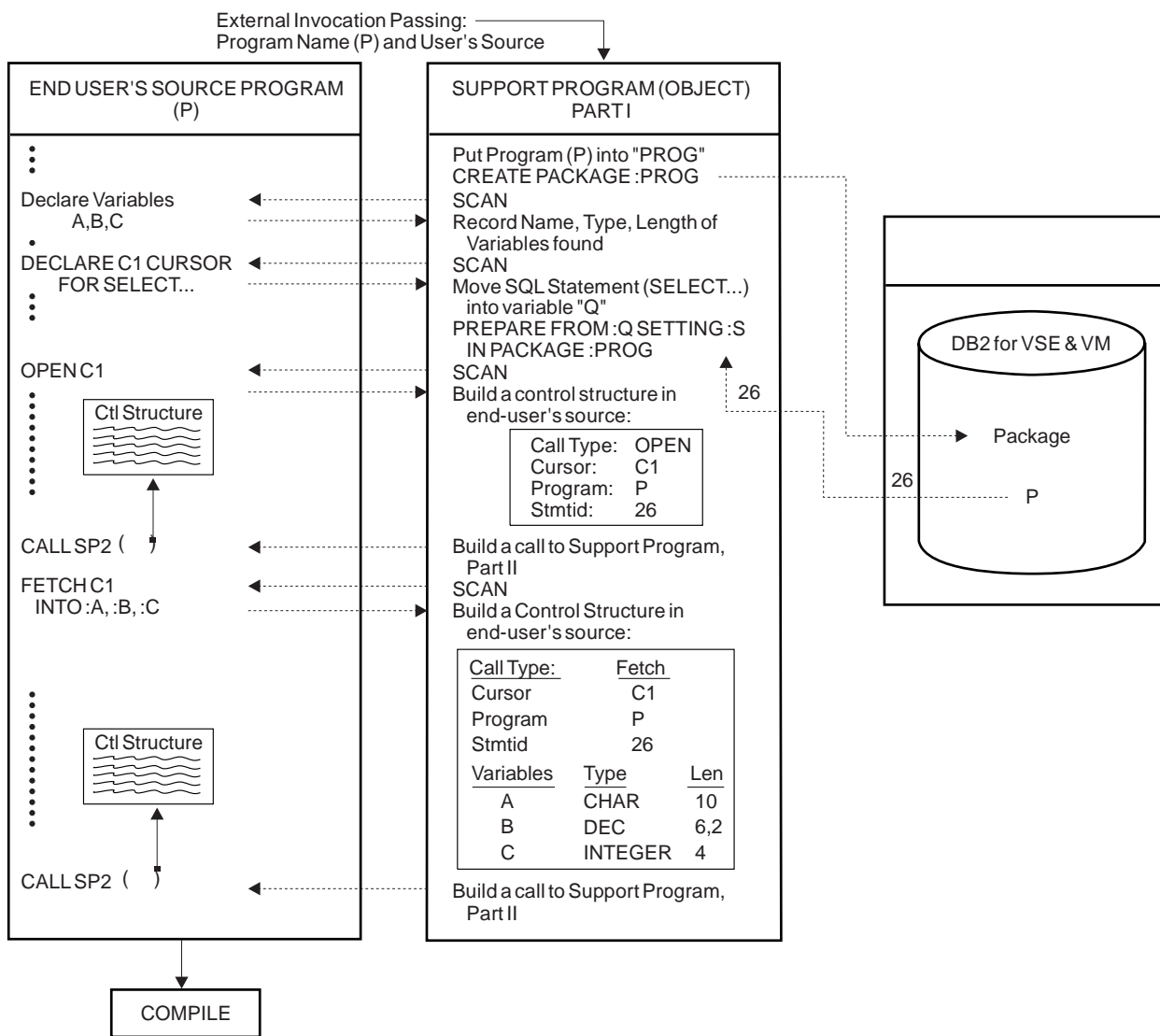


Figure 68. Pseudocode Example of Preprocessing the End-User Program P

Figure 69 on page 193 shows the execution-time flow between the end-user's object program and the support program (Part 2) in more detail. The two calls shown correspond to the two calls generated in Figure 68. This example does not

go far enough to illustrate that two calls of the same type (two opens, for example) would share the same set of logic and the same extended dynamic statement (OPEN) in the support program.

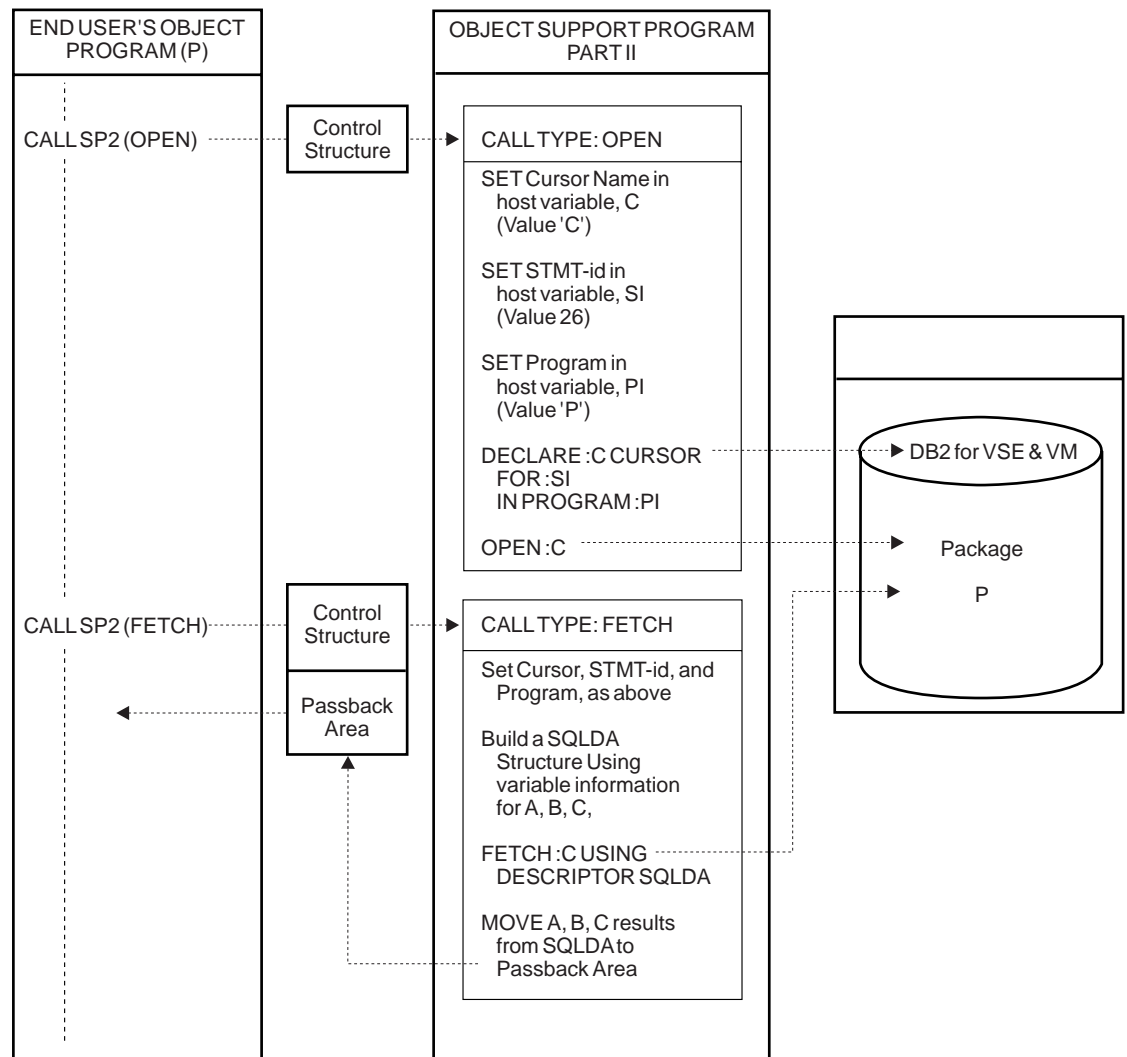


Figure 69. Pseudocode Example of Executing the End-User Program P

Grouping Extended Dynamic Statements in an LUW

There are primarily three cases to consider when determining the proper grouping of extended dynamic statements in a logical unit of work:

1. An LUW contains a CREATE PACKAGE without the MODIFY option. This would be the case for a language preprocessor application.
2. An LUW contains a CREATE PACKAGE with the MODIFY option. This would be the case for an application that gets new SQL statements from its users, then prepares and executes them immediately (but also has them available for later execution, because they are stored in a package).
3. An LUW contains no CREATE PACKAGE (the referenced package has been created with the MODIFY option in another LUW). This would be the case for

an application that prepares, executes, or changes statements in a package that was created previously.

In the first case, the only other extended dynamic statement permitted is the PREPARE statement, and it must reference only the program that is specified in the CREATE PACKAGE statement. If the LUW is terminated by a COMMIT statement, a DB2 Server for VSE package is created. If no extended PREPARE statements were executed, the package is empty and the COMMIT statement returns an SQLCODE of -759 (SQLSTATE '42943'). If a ROLLBACK statement terminates the LUW, no package is created. In Figure 70, Example 1 is a valid illustration of this case.

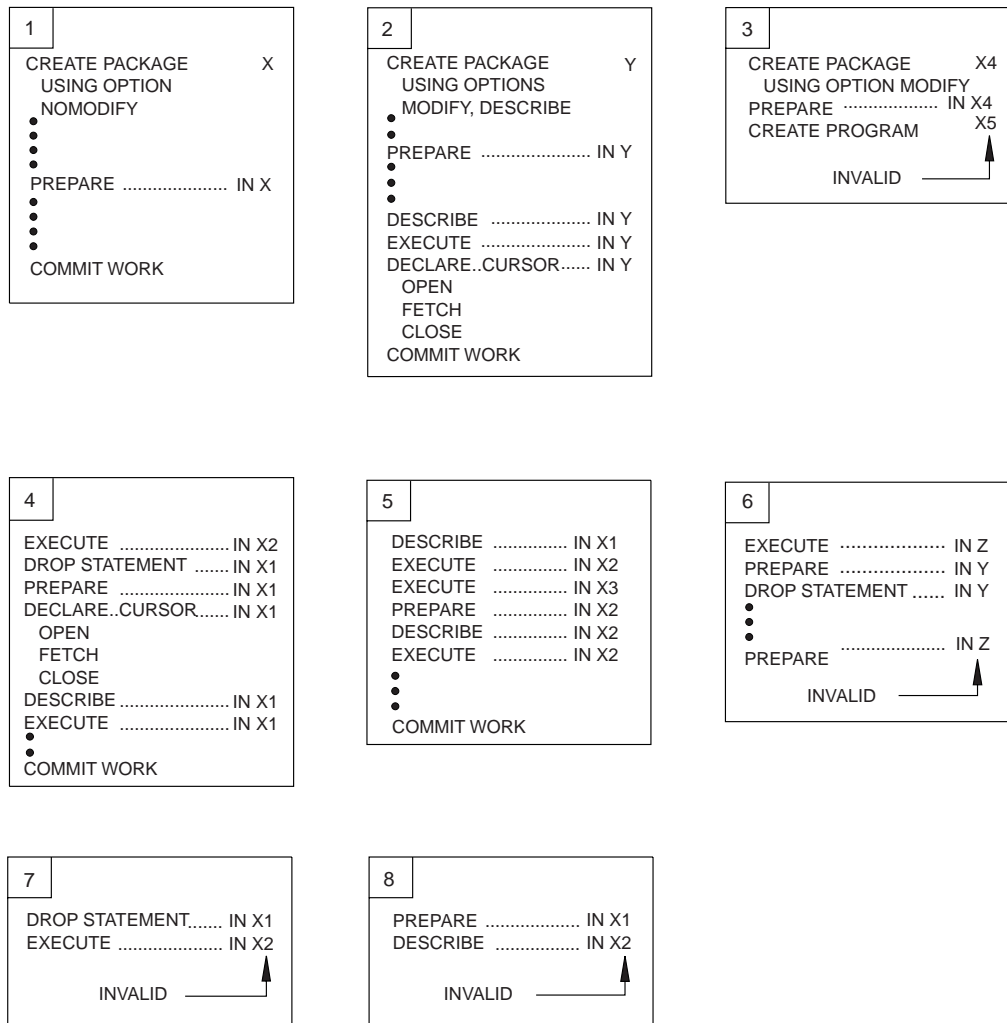


Figure 70. Placement of Extended Dynamic Statements in Logical Units of Work

In the second case, the rules discussed above for case 1 apply, but Extended DESCRIBE, EXECUTE, DECLARE CURSOR, OPEN, FETCH, DROP STATEMENT, and CLOSE statements may also be used in the same LUW, referencing the statements just added to or already contained in the current package. However, you cannot reference a package other than the one created in the current LUW. In Figure 70, example 2 is a valid example of this case. Example 3 illustrates an invalid case 2 sequence. If the current LUW is committed before extended PREPAREs are used to add statements to it (it is empty), it still may be

extended in a later LUW (since it is modifiable, it may make sense to leave it empty initially).

In case 3, where the current LUW contains no CREATE PACKAGE, extended dynamic statements may reference any package that has been created with a CREATE PACKAGE statement. However, after an extended dynamic statement that causes modification of the package is used (an extended PREPARE or DROP STATEMENT), subsequent extended dynamic statements in the same LUW may only refer to the modified package. Once the LUW is terminated, reference to any package that has been created by a CREATE PACKAGE may be resumed. (Note that this does not preclude additional restrictions: to modify a package, you must have created it with the MODIFY option, and to DESCRIBE a statement in a package, it must have been created with the DESCRIBE option.)

For example, if packages X1, X2, and X3 have been created with a CREATE PACKAGE, where X1 and X2 have the MODIFY and DESCRIBE options. Examples 1, 2, 4, and 5 in Figure 70 on page 194 are valid, while Examples 3, 6, 7, and 8 are invalid.

Using COMMIT WORK and ROLLBACK WORK Statements

It is a good practice to always do a COMMIT WORK or ROLLBACK WORK in your program that contains extended dynamic statements before you terminate the program. If you use extended dynamic statements in a CICS/VSE transaction, it is imperative that a COMMIT or ROLLBACK WORK be done before ending the transaction. (A CICS abnormal termination may occur, especially if the logical unit of work contains a CREATE PACKAGE statement.)

Considering Virtual Storage in a Logical Unit of Work

If virtual storage consumption by the database manager is an important consideration, you should be aware of the trade-off in using modifiable packages. The amount of virtual storage required to represent statements prepared in the current LUW may be significantly more than that required for previously prepared statements. If you enter a COMMIT WORK before executing the statement, the virtual storage requirement for the package will be considerably less, but additional work will be performed to store the updated package and to reload it for execution.

You should make this trade-off based on the nature of the preprocessing in your application.

When declaring extended dynamic cursors, you must consider virtual storage requirements. Cursor names are dynamically mapped to statement numbers when DECLARE CURSOR statements are executed. A small amount of virtual storage is required for each uniquely named cursor declared in an LUW. This storage is not released until the end of the LUW. The amount of storage held, therefore, can become quite large when many unique cursor names are declared.

Mapping Extended Dynamic Statements to Static and Dynamic Statements

Figure 71 shows how static and dynamic SQL statements are mapped to the SQL statements that preprocess and execute them.

<i>Figure 71. Mapping Extended Dynamic to Static and Dynamic Statements</i>		
Static and Dynamic SQL Statement	SQL Statement Executed at Preprocessing Time	SQL Statement Executed at Run Time
CLOSE	N/A	Extended CLOSE
COMMIT	N/A	COMMIT
CONNECT	N/A	CONNECT
DECLARE CURSOR FOR <i>statement</i>	Basic Extended PREPARE of <i>statement</i>	Extended DECLARE CURSOR
DECLARE CURSOR FOR <i>statement_name</i>	See Figure 72	Extended DECLARE CURSOR
DESCRIBE <i>statement_name</i>	N/A	Extended DESCRIBE
DROP PACKAGE	N/A	DROP PACKAGE
EXECUTE	N/A	Extended EXECUTE
EXECUTE IMMEDIATE <i>string_constant</i>	Basic Extended PREPARE of <i>string_constant</i>	Extended EXECUTE
EXECUTE IMMEDIATE <i>host_variable</i>	Empty Extended PREPARE	Temporary Extended PREPARE Extended EXECUTE
FETCH	N/A	Extended FETCH
OPEN	N/A	Extended OPEN
PREPARE <i>string_constant</i>	Basic Extended PREPARE of <i>string_constant</i> ¹	N/A
PREPARE <i>host_variable</i>	Empty Extended PREPARE ¹	Temporary Extended PREPARE
PUT	N/A	Extended PUT
ROLLBACK	N/A	ROLLBACK
SELECT INTO	Single row Extended PREPARE	Extended EXECUTE
Other executable statements	Basic Extended PREPARE	Extended EXECUTE
Non-executable statements	N/A	N/A
Note:		
1. See Figure 72 if used in context of a cursor.		

Figure 72 shows the SQL statements that prepare statements executed with a cursor.

<i>Figure 72. Preprocessing Related PREPARE and DECLARE CURSOR Statements</i>	
Example Statements	Extended Dynamic SQL Statement Executed at Preprocessing Time
PREPARE <i>string_constant</i> DECLARE CURSOR <i>statement_name</i>	Basic Extended PREPARE N/A
PREPARE <i>host_variable</i> DECLARE CURSOR <i>statement_name</i>	Empty Extended PREPARE N/A
DECLARE CURSOR <i>statement_name</i> PREPARE <i>string_constant</i>	Empty Extended PREPARE Temporary Extended PREPARE ¹
DECLARE CURSOR <i>statement_name</i> PREPARE <i>host_variable</i>	Empty Extended PREPARE N/A
Note:	
1. This example is not supported in packages created with the NOMODIFY option specified.	

SQL Functions Not Supported in Extended Dynamic Statements

The following SQL facilities are not supported for statements that are prepared using extended dynamic SQL, unless the application program that performs the extended PREPARE statement supplies the support:

- Checking of the statement for conformance to SQL-89 or SAA standards
- Use of SQL comments
- Optional choice for the FOR UPDATE OF clause in cursor query statements
- Use of negative indicator values in predicates, unless the statement is prepared using the descriptor format of the extended PREPARE statement.

These restrictions do not apply to FORTRAN application programs, because the DB2 Server for VSE preprocessors provide the necessary support.

Refer to the *DB2 Server for VSE & VM SQL Reference* manual for more information on restrictions that apply to extended dynamic statements.

Chapter 8. Maintaining Objects Used by a Program

Contents

Managing Dbspaces	200
Defining Dbspaces	200
Finding Available Space	202
Specifying Properties of Dbspaces	202
Modifying the Size of Dbspaces	204
Automatically Locking Dbspaces	204
Overriding Automatic Locking	205
Deleting the Contents of Dbspaces	206
Other Data Definition Statements	206
Using Tables, Indexes, Statistics, Synonyms, Comments, and Labels	207
Creating Tables	207
Modifying Tables	208
Dropping Tables	208
Using Indexes	208
Updating Catalog Tables for Table and Index Activity	208
Using Synonyms	209
Using Comments	209
Using Labels	209
Using Stored Procedures and PSERVERS	209
Using Stored Procedures	209
Example of a Stored Procedure Definition	210
Using PSERVERs	211
Example of a Stored Procedure Server Definition	212

Managing Dbspaces

This section discusses the data definition statements for dbspaces and should be read in conjunction with the *DB2 Server for VSE & VM SQL Reference* manual, which contains the syntax, authorization rules, and usage rules of these statements.

Note: This section applies to DB2 Server for VSE & VM application servers only.

Defining Dbspaces

A *dbspace* is a portion of the database that can contain one or more tables and any associated indexes. Each table that is stored is placed in a dbspace chosen by the creator of the table.

Dbspaces are defined when the database is generated and may be added later by the ADD DBSPACE process. Each dbspace remains unnamed and *available* until it is *acquired* with an ACQUIRE DBSPACE statement, generally by the Database Administrator (DBA). An acquired dbspace can be later returned to the list of available dbspaces by the DROP DBSPACE statement.

The user who acquires a dbspace can either specify from which *storage pool* the database manager is to acquire the dbspace, or can allow the system to choose the storage pool by default. Storage pool are collections of data sets called dbextents, and control the distribution of the database across direct access storage devices (DASD).

Storage pools can be *recoverable* or *nonrecoverable*. Recoverable storage pools protect their data using the automatic recovery for data updates. With nonrecoverable storage pools, system overhead is reduced, but if there is a system failure, some data may be lost, because the burden of recovery is placed on the user. Nonrecoverable storage pools are particularly useful in cases where large amounts of data are loaded from an external source, and that data is never modified thereafter. See the *DB2 Server for VSE System Administration* manual for more information about storage pools.

The acquiring user also gives a name to the dbspace, and defines certain characteristics for it. If it is to be private, the user who acquires it becomes its *owner*; if it is of type public, its owner becomes public.

If you have DBA authority, you can acquire a dbspace for another user by concatenating the *userid* to the dbspace-name:

```
ACQUIRE PRIVATE dbspace NAMED JONES.SPACE1
```

In the above statement, the owner of the dbspace is user JONES. User JONES can refer to the dbspace as simply SPACE1.

A user holding RESOURCE authority can create new tables in any public dbspace, or in any private dbspace owned by that user. Users who do not have RESOURCE authority can also create tables in any private dbspace that was acquired for that user by the DBA. Only users having DBA authority can create tables in a private dbspace owned by another user.

The ability to access and update tables belonging to another user is controlled by the system. Authorized users can access and update tables in any dbspace of any type, by adding the owner-name as a prefix to the table name (for example, SMITH.INVENTORY).

Note: Even users who are authorized to access data in someone else's dbspace may not be permitted to do so if the dbspace is in use.

An attempt to *read* data in a private dbspace results in a negative SQLCODE if any data in the dbspace has been *modified* by a still-active logical unit of work. An attempt to *modify* data in a private dbspace results in a negative SQLCODE if any data in the dbspace has been *read or modified* by a still-active logical unit of work. If the locked data you attempt to access is in a public dbspace, your program waits and does not regain control until the lock is freed. If you attempt to update locked data in a private dbspace, the system immediately returns control to your program, with a negative SQLCODE.

The size of the space that is locked is the *lock size*. The lock size on a private dbspace is always the entire dbspace, while the default lock size on a public dbspace is somewhat smaller to allow for more concurrency. Thus, you should place tables in public dbspaces if you expect that more than one user may need concurrent access to them. On the other hand, because operations on private dbspaces do not pay the overhead of acquiring individual locks within the dbspace, a private dbspace is an efficient place to store tables for the exclusive use by one user at a time. The cost of smaller locks is higher overhead. Figure 73 and Figure 74 summarize the database manager locking mechanism.

Refer to the *DB2 Server for VSE Diagnosis Guide and Reference* manual for more information on locking.

Figure 73. Locking Summary for Private Dbspaces

If you attempt to:	But another user has already:	
Read data	read the data (acquired a share lock)	modified the data (acquired an exclusive lock)
Modify data	You are allowed to read the data	You receive a negative SQLCODE
	You receive a negative SQLCODE	You receive a negative SQLCODE

The lock size for a private dbspace is always the entire dbspace.

Figure 74. Locking Summary for Public Dbspaces

If you attempt to:	But another user has already:	
Read data	read the data (acquired a share lock)	modified the data (acquired an exclusive lock)
Modify data	You are allowed to read the data	Your program waits
	Your program waits	Your program waits

The lock size of a public dbspace defaults to a page (4096 bytes). The lock size can be changed by the ACQUIRE DBSPACE or ALTER DBSPACE statements.

Finding Available Space

The ACQUIRE DBSPACE statement causes the system to find an available dbspace of the requested type (public or private) and give it the dbspace-name you specify. The dbspace-name must be an SQL identifier, as described in the *DB2 Server for VSE & VM SQL Reference* manual; you can use it to refer to the DBSPACE in other SQL statements, such as CREATE TABLE.

If the dbspace type is public, its owner becomes public; if the type is private, its owner becomes the user who preprocessed the program in which the ACQUIRE DBSPACE is embedded. Dbspace names must be unique within all the dbspaces owned by the same user, but may duplicate the name of a dbspace owned by another user.

Specifying Properties of Dbspaces

You can optionally specify one or more of the following properties of a dbspace, in any order. Separate the parameters with commas.

NHEADER Number of Header Pages. The number of 4096-byte logical pages in the dbspace that are reserved for header pages. The system uses header pages to record information about the contents of the dbspace.

Notes:

1. NHEADER cannot be larger than eight pages.
2. If NHEADER is not specified, the default is eight pages.
3. You cannot change NHEADER after the dbspace has been acquired. If you choose a small number for NHEADER, it may limit the number of tables that can be created in the dbspace.

PAGES Number of Pages. The minimum number of 4096-byte logical pages that you require for this dbspace.

Notes:

1. The system may actually give you more pages than you request because it acquires storage in units of 128 pages. However, of the available dbspaces, the one chosen will be the *smallest* that will satisfy the size specified for PAGES. The system determines the number of pages that you receive by rounding the number you specify to the next higher multiple of 128 pages. For example, if you specify PAGES=53, the system acquires a block of 128 pages. If you specify PAGES=130, the system acquires 256 pages.
2. If you do not specify PAGES, the system acquires the smallest available dbspace by default.

PCTINDEX Percentage of Index Pages. The percentage (0 to 99) of *all* pages in the dbspace that are reserved for indexes.

Notes:

1. If you do not specify PCTINDEX, the default is 33 percent.
2. You cannot change PCTINDEX after the dbspace has been acquired. If you choose a small number for PCTINDEX, it may limit the number of indexes that can be created on tables in the dbspace. (If you find that the PCTINDEX is too small, you can acquire another dbspace and move the data there.)

PCTFREE Percentage of Free Space. The percentage (0 to 99) of the space on *each* page that the system is to keep empty when data is inserted into the dbspace.

Notes:

1. If you do not specify PCTFREE, the default is 15 percent.
2. Typically a user might acquire a dbspace with PCTFREE set to some value such as 25 percent. The dbspace is then loaded with data by the Database Services Utility (described in the *DB2 Server for VSE & VM Database Services Utility* manual). The system ensures that at least 25 percent of the space on each page is left empty. After the initial loading of the dbspace, the user can set PCTFREE to zero by means of the ALTER DBSPACE statement (described later). Then, in subsequent insertions, the system places new data in the space reserved during initial loading. Using reserved free space in this way results in a more favorable physical clustering of data on pages when the data is loaded, and, therefore, improves access time. The *DB2 Server for VSE Database Administration* manual discusses data clustering in more detail.
3. The value of PCTFREE is critical during mass insertion of data into a dbspace (for example, a DBS Utility DATALOAD command). Refer to the appendix on estimating the number of data pages required in the *DB2 Server for VSE Database Administration* manual for more information on the dbspace percent free specification.

LOCK Lock Size. Applicable to public dspsaces only (private always locks a dbspace). The valid specifications for *size* are DBSPACE, PAGE, and ROW.

Notes:

1. The lock size determines the size of the locks that are acquired when a user reads or updates data. If you specify ROW, the system locks only an individual row in the table; PAGE or DBSPACE cause the smallest lockable unit to be a page (4096 bytes) or a dbspace, respectively. Key-level locking is used for indexes on tables in dspsaces for which row-level locking is specified.
2. In general, using larger locking units causes less overhead to be spent in acquiring locks, but also limits concurrency.
3. The default lock size for each public dspspace is PAGE.

STORPOOL Storage Pool Number. Indicates from which storage pool a dbspace is to be acquired.

Notes:

1. If a dbspace of the specified type and size is not available in this storage pool, the ACQUIRE DBSPACE is unsuccessful, and a negative SQLCODE is returned.
2. If you do not specify STORPOOL, the system acquires a dbspace of the correct type and size from *any recoverable* storage pool. To acquire a dbspace from a nonrecoverable storage pool, you *must* specify the STORPOOL parameter.

Modifying the Size of Dbspaces

The ALTER DBSPACE statement enables you to alter the percentage of free space that is reserved on each data page when records are inserted into a public or private dbspace. It also enables you to alter the lock size of a public dbspace. (You cannot alter the lock size of a private dbspace.)

When you acquire a dbspace, you should set the percentage (0 to 99) of free space to some number greater than zero (the default is 15 percent). A typical use of ALTER DBSPACE is to set the percentage of free space to zero (PCTFREE=0) after initial loading of data into a dbspace; subsequent insertions can then take advantage of the free space that is reserved during the loading process. It is also possible to increase PCTFREE again for a later loading phase.

To alter the lock size of a public dbspace at any time, use the LOCK parameter. (You can specify both the PCTFREE and LOCK parameters when altering a public dbspace, in either order, separated with a comma. Each may be specified only once.) The valid lock sizes are ROW, PAGE, and DBSPACE, as described under the ACQUIRE DBSPACE statement. When an ALTER DBSPACE statement is executed to alter the lock size of a dbspace, the system acquires an exclusive lock on the entire dbspace and holds the lock until the end of the current logical unit of work. The newly selected lock size then becomes effective for subsequent logical units of work.

Automatically Locking Dbspaces

When you operate the database manager in single user mode, there is no contention from other users when you attempt to access data; there may be however in multiple user mode. To provide for concurrent access, the system internally acquires locks on data accessed by a logical unit of work.

All LUWs automatically acquire *exclusive locks* on all data that they modify, and *share locks* on data that they are reading. Exclusive locks prevent other users from either reading or modifying the data; share locks permit other users to read, but prevent them from modifying the data.

For UPDATE and DELETE processing, the system acquires *update locks*. If the user wants to change the data, the *update lock* is changed to an *exclusive lock*; otherwise, the *update lock* is changed to a *share lock*. An *update lock* is acquired for a *Positioned* DELETE only if the cursor was declared with the FOR UPDATE clause. This type of lock is also acquired on a parent table when changes are made to its dependent tables. In general, locks are held to the end of the LUW in

which they are acquired. (See “Selecting the Isolation Level to Lock Data” on page 124 for more information.)

Potential deadlocks are automatically detected and corrected. A deadlock occurs when two LUWs are each waiting to access data that the other has locked. The system detects this situation and backs out the most recent LUW, meaning that all changes made to the database during the LUW are restored, and then the locks that were acquired for the LUW are released. The other application can then proceed. If your LUW is backed out, a negative SQLCODE is returned and SQLWARN6 is set to W.

Locking is automatic and requires no user intervention. However, certain statements permit users to adjust or override the normal locking. You can adjust the size of the lockable data units with the LOCK option of the ACQUIRE DBSPACE and ALTER DBSPACE statements. You can also override automatic locking and explicitly acquire certain kinds of locks with the LOCK statement as discussed below.

Note: Only single user mode prevents locking.

Overriding Automatic Locking

The LOCK statement overrides the automatic locking mechanism and explicitly acquires a lock on a table or dbspace, which is held the end of the current LUW.

The LOCK statement is useful only in multiple user mode. In single user mode, there is no contention for resources, and, hence, no locking. When running in single user mode, all LOCK statements are ignored.

An *exclusive* lock prevents other users from either reading or changing any data in the locked table or dbspace. A *share* lock permits other users to read, but prevents them from modifying, the data in the locked object.

The requested lock may be unavailable because other LUWs are reading or modifying the indicated data. If this is the case, the LUW that requested the lock waits until the other active LUWs have ended. The system then grants the lock, and the requesting LUW proceeds normally.

The LOCK statement is entirely optional, as the system has fully automatic locking. You may issue all SQL queries and updates independently of explicit LOCK statements.

The LOCK statement is useful mainly for avoiding the overhead of acquiring many small locks when scanning over a table. For example, suppose some dbspace has been acquired with a lock size of ROW. If you know that you will be accessing all the rows of a table within that dbspace, you may want to explicitly lock the entire table to avoid the overhead of acquiring locks on each individual row.

In a private dbspace, a LOCK statement on a table is the same as one on the entire dbspace, because locking is always done at the DBSPACE level for private dbspaces.

Deleting the Contents of Dbspaces

The DROP DBSPACE statement deletes the entire contents of a dspace. When the logical unit of work is committed, the dspace is available to be acquired. The DROP DBSPACE statement is a much faster way to delete the contents of a dspace than by deleting the data one row at a time or dropping one table at a time. (You can use DROP DBSPACE with both public and private dspaces.)

For any table that is dropped implicitly by the DROP DBSPACE statement, all referential constraints in which it is a dependent are dropped, and all referential constraints in which it is a parent are also dropped. Furthermore, any unique constraints defined in the table are dropped.

When a dspace is dropped, packages for programs that operate on that dspace are marked *invalid*. In addition, if a parent table has been dropped, the packages with tables dependent on that parent table are also marked invalid, because the relationship between the parent table and its dependent tables was dropped.

If one of these programs is running, the system does not drop the dspace until the running program ends its current LUW. The invalid packages remain in the database until they are explicitly dropped using the DROP PACKAGE statement (discussed in the *DB2 Server for VSE & VM SQL Reference* manual).

When an invalid package is invoked, the system attempts to dynamically re-preprocess it. If the package was **not** invalidated because the relationship between a parent table and its dependent tables was dropped, and the program contains any SQL statement that refers to a dspace or table that has been dropped, that SQL statement returns a negative SQLCODE at execution time.

Other Data Definition Statements

In addition to SQL data definition statements for dspaces, there are those that enable you to:

- Create and drop tables (CREATE TABLE and DROP TABLE)
- Create and drop indexes on tables (CREATE INDEX and DROP INDEX)
- Add new columns to existing tables; and add, drop, activate, or deactivate primary keys, foreign keys, and unique constraints (ALTER TABLE)
- Create and drop synonyms for table names (CREATE SYNONYM and DROP SYNONYM)
- Enter comments about tables into the DB2 Server for VSE catalog tables (COMMENT ON)
- Label tables and columns in dynamic SQL application programs (LABEL ON).

The following discussion is only an introduction to these statements. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for their syntax and detailed usage rules.

Using Tables, Indexes, Statistics, Synonyms, Comments, and Labels

One advantage of the database manager is that you can define new objects in the database without stopping the system or invoking special utilities. This provides great flexibility: for example, your application program can create a table for storing and manipulating some temporary result, and drop the table when it is no longer needed.

Data definition statements automatically update the catalog tables that describe the database. (These catalog tables are explained in the *DB2 Server for VSE & VM SQL Reference* manual.) If an error occurs while you are processing a data definition statement, the system stops processing the statement, and reverses only the changes resulting from the statement in error. Any work done before the execution of the statement in the LUW will not be affected. If you want to, you can enter a ROLLBACK statement to undo any other changes made in the LUW.

Also, if you plan to DROP and re-CREATE the object later in the program, make sure that you start a new LUW after you drop the object. For example, if you write a procedure that creates and drops a temporary table, make sure that your program issues a COMMIT before the end of the procedure. (For more information on the LUW refer to “Using Logical Units of Work” on page 14.)

Some data definition statements may invalidate the packages of one or more programs previously preprocessed. For example, dropping the index used by a program to access a table will invalidate the package of that program. Other examples include adding keys (primary or foreign) to a table, or dropping, activating, or deactivating keys on the table. When the program is used, a new package is created based on the dependencies currently available. No changes need be made to the program. The process of creating the new package called rebinding is entirely transparent to programs, except for a slight delay in processing the first SQL statement. (Rebinding is discussed in Chapter 4, “Preprocessing and Running a Program” on page 105.)

Creating Tables

Use the CREATE TABLE statement to create a new table in the database and to define the datatypes and subtypes of all the columns in the table. You can also use it to define primary keys and foreign keys which may be used to ensure referential integrity. This is done by specifying a primary key, a foreign key, and a delete or update rule that defines the relationship. Only a primary key is required for entity integrity.

If you specify the NOT NULL option for a column, the system does not permit null values in that column. Any statement that attempts to place a null value in such a column is rejected with an error code.

You can also associate a field procedure with a column. For more information on field procedures see “Using Field Procedures” on page 225.

You can define a *unique constraint* when creating a table. This consists of one or more columns where the combined value in these columns is unique. This enables you to ensure data integrity for columns where a primary key would not be practical.

Note: Instead of declaring a column to be of DECIMAL (or NUMERIC) data type with a scale of 0, you should consider declaring it INTEGER or SMALLINT.

These data types use storage more effectively, and other processing will be more efficient. If the precision is less than 5, use SMALLINT; if the precision ranges from 5 to 7, use INTEGER.

Once a table has been created, you may not change the data types of its columns or drop a column from the table. However, you may add new columns, a primary key, foreign keys, and unique constraints by using the ALTER TABLE statement.

Modifying Tables

Use the ALTER TABLE statement to add a new column to an existing table, or to add, drop, activate or deactivate primary keys, foreign keys, and unique constraints.

Dropping Tables

Use the DROP TABLE statement to drop a table from the database. All indexes, primary and foreign keys, unique constraints, views defined on the table, and all privileges granted on the table, are also dropped. All contents of the table are lost. However, users can have previously defined synonyms (by a CREATE SYNONYM statement) for the name of the table that was dropped; these synonyms remain in effect even though the table no longer exists.

Using Indexes

Use the CREATE INDEX statement to create an index on one or more columns of a table, and to give a name to the new index. The indicated table must exist, but it may be empty.

You can create an index on a column in either ascending (ASC) or descending (DESC) order. Ascending order is the default. Performance may be improved for queries that access the indexed column in the specified order.

An index is maintained until it is explicitly dropped with a DROP INDEX statement, or until its table or dbspace is dropped.

Indexes are invisible to application programs in the sense that the system provides no means for using an index directly. The database manager selects the index, if any, that is to be used in processing a given query or data manipulation statement.

Updating Catalog Tables for Table and Index Activity

Use the UPDATE STATISTICS statement to bring up to date the internal statistics recorded by the system for a table and its indexes. These statistics, which are contained in the catalog tables, include the size of the table, various index characteristics, and other information. The system uses these statistics when choosing access paths for SQL statements. If the statistics are not kept up to date, less efficient access paths may be chosen.

You should invoke the UPDATE STATISTICS statement for a table after a significant number of changes have been made to its data since it updated; for example, if a table has been changed by 20 percent or more.

Using Synonyms

Use the CREATE SYNONYM statement to define an alternative name for a table or view. For example, the following statement defines the alternative name PEOPLE to refer to the table named EMPLOYEE whose owner is SMITH:

```
CREATE SYNONYM PEOPLE FOR SMITH.EMPLOYEE
```

The right-hand side of the CREATE SYNONYM statement (SMITH.EMPLOYEE in the above example) must be the name of a table or a view, not another synonym.

Synonyms are commonly used when a group of users all want to share a table. Suppose one user, ADAMS, creates a table called DATA. All users sharing this table can then enter the statement:

```
CREATE SYNONYM DATA FOR ADAMS.DATA
```

Each user can then refer to the shared table as DATA, without using the fully qualified name ADAMS.DATA. (Remember that ADAMS must authorize the other users to access his table.)

Once created, a synonym remains in effect until it is explicitly dropped by a DROP SYNONYM statement.

Using Comments

Use the SQL COMMENT ON statement to associate remarks or comments with your tables or views, or with columns in your tables or views. The comment you specify is placed into one of the catalog tables.

Using Labels

Use the SQL LABEL ON statement to define a label for a table name or a column name. Unlike synonyms, labels cannot be used as identifiers. Instead, they can be used in displays created by applications that process SQL statements dynamically. You can enter SQL statements using the actual table and column names (which are easier to enter). The program can display the results using the labels (which are easier to understand) instead of the table and column names.

Labels are ignored by DBS Utility and ISQL SELECT processing. Only column names will identify SQL *select-statement* output displayed by DBS Utility or ISQL processing.

Using Stored Procedures and PSERVERS

Using Stored Procedures

Before a stored procedure can run, you must define it to DB2. Use the SQL statement CREATE PROCEDURE to define a stored procedure to DB2. To alter the definition, use the ALTER PROCEDURE statement.

Figure 75 on page 210 lists the characteristics of a stored procedure and the CREATE PROCEDURE and ALTER PROCEDURE parameters that correspond to those characteristics.

<i>Figure 75. Characteristics of a Stored Procedure</i>	
Characteristic	CREATE/ALTER PROCEDURE Parameter
Stored procedure name Parameter declarations	PROCEDURE
External name	EXTERNAL NAME
Language	LANGUAGE ASSEMBLE LANGUAGE C LANGUAGE COBOL LANGUAGE PLI
Parameter style	PARAMETER STYLE GENERAL PARAMETER STYLE GENERAL WITH NULLS
Name of group of servers where stored procedure can run	SERVER GROUP <i>server-group-name</i>
Whether or not a stored procedure can run in default server group	DEFAULT SERVER GROUP YES DEFAULT SERVER GROUP NO
Load module stays in memory after it executes	STAY RESIDENT NO STAY RESIDENT YES
Run-time options	RUN OPTIONS <i>options</i>
Maximum number of result sets returned	RESULT SETS <i>integer</i>
Commit work on return from stored procedure	COMMIT ON RETURN YES COMMIT ON RETURN NO

For information on the parameters for the CREATE PROCEDURE or ALTER PROCEDURE statement, see the *DB2 Server for VSE & VM SQL Reference manual*.

Example of a Stored Procedure Definition

Suppose you have written and prepared a stored procedure that has these characteristics:

- The name is B.
- It takes two parameters:
 - An integer input parameter named V1
 - A character output parameter of length 9 named V2
- It is written in the C language.
- The load module name is SUMMOD.
- The parameters can have null values.
- It should be deleted from memory when it completes.
- The Language Environment run-time options it needs are:


```
MSGFILE(OUTFILE) ,RPTSTG(ON) ,RPTOPTS(ON)
```
- It can be executed by any stored procedure server in the group named PAYROLL.
- It can return at most 10 result sets.

- When control returns to the client program, DB2 should not commit updates automatically

This CREATE PROCEDURE statement defines the stored procedure to DB2:

```
CREATE PROCEDURE B(V1 INTEGER IN, V2 CHAR(9) OUT)
LANGUAGE C
EXTERNAL NAME SUMMOD
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON) '
SERVER GROUP PAYROLL
DEFAULT SERVER GROUP NO
RESULT SETS 10
COMMIT ON RETURN NO;
```

Later, you need to make the following changes to the stored procedure definition:

- The stored procedure can also be run in the default server group in addition to the group of stored procedure servers named PAYROLL.

Execute this ALTER PROCEDURE statement to make the changes:

```
ALTER PROCEDURE B
DEFAULT SERVER GROUP YES;
```

Using PSERVERS

Stored procedures are executed by stored procedure servers. These servers are organized into named groups. Use the SQL statement CREATE PSERVER to add a stored procedure server to a group. To alter the definition, use the ALTER PSERVER statement.

Figure 76 lists the characteristics of a stored procedure server and the CREATE PSERVER and ALTER PSERVER parameters that correspond to those characteristics.

Figure 76. Characteristics of a Stored Procedure

Characteristic	CREATE/ALTER PSERVER Parameter
Stored Procedure server name	PSERVER <i>procedure-server</i>
Name of the group to which the stored procedure server belongs	GROUP <i>group-name</i>
Whether or not the database manager should issue a START PSERVER command when the database initializes	AUTOSTART NO AUTOSTART YES
A description of the stored procedure server	DESCRIPTION <i>description</i>

For information on the parameters for the CREATE PROCEDURE or ALTER PROCEDURE statement, see the *DB2 Server for VSE & VM SQL Reference* manual.

Example of a Stored Procedure Server Definition

Suppose you must set up a stored procedure server that has these characteristics:

- The name is SERVER1
- It is part of stored procedure group PAYROLL
- The database manager is not to issue a START PSERVER command when it initializes

This CREATE PSERVER statement defines the stored procedure server to DB2:

```
CREATE PSERVER SERVER1
GROUP PAYROLL
AUTOSTART NO
```

Later, you need to make the following changes to the stored procedure server definition:

- The database manager should issue a START PSERVER command when it initializes
- The description of the stored procedure server is to be "This is the first server used by payroll procedures"

Execute this ALTER PROCEDURE statement to make the changes:

```
ALTER PSERVER SERVER1
AUTOSTART YES
DESCRIPTION 'This is the first server used by payroll procedures'
```

Chapter 9. Assigning Authority and Privileges

Contents

Defining User Access to the Database	214
Defining Authority Types for the Database	214
Granting Authority to Users	214
Revoking Authority from Users	214
Defining Privileges	215
Defining Privileges on Tables and Views	215
Revoking Privileges	216
Defining Privileges on Packages	216
Assigning User Privileges to the Owner	216
Assigning Privileges to Others	217
Determining When the Owner Can Grant the RUN Privilege	217
Differences Between Static and Dynamic Statements	218
Revoking the Run Privilege	218
Recording Assigned Privileges in the Catalog Tables	219

Defining User Access to the Database

Defining Authority Types for the Database

When a database is initially generated, there is only one user defined for it. This user, referred to as SQLDBA, has a special authority called “DBA” authority. Only someone with DBA authority can grant authorities to other users.

The four types of authorities are:

CONNECT	Authorization to access the database
RESOURCE	Authorization to acquire space in the database
SCHEDULE	Authorization to issue a connect without a password (internal to the on-line Resource Adapter)
DBA	Authorization to perform database administration functions.

Granting any one of these authorities to a user who does not already have the CONNECT authority causes that user to be granted CONNECT authority. For example, if resource authority is granted to a user who currently has no authorities, the user will have both RESOURCE and CONNECT authority; if DBA authority is granted, the user will have DBA, CONNECT, SCHEDULE, and RESOURCE authorities.

Granting Authority to Users

Granting an authority that a user already possesses has no additional effect, except for changing the password if it is specified.

You should not grant CONNECT authority to SYSTEM or PUBLIC. They are used internally.

Revoking Authority from Users

Note: In discussions about revoking authorities and privileges in this chapter, the “revoker” is defined as the user who preprocessed the program in which the REVOKE statement appears. However, for dynamically defined REVOKE statements, the revoker is determined at run time, based on the connected authorization ID.

The System Authorities form of the REVOKE statement allows a *user having DBA authority* to revoke an authority from any other users regardless of who originally granted it. The only exceptions are:

- Anyone with DBA authority cannot revoke their authority
- No one can revoke RESOURCE authority from a user who has DBA authority.

See the *DB2 Server for VSE & VM SQL Reference* manual for the syntax of the REVOKE statement.

If you enter REVOKE for an authority that a user does not have, the revocation is ignored.

Revoking a user's CONNECT authority causes any other authorities to be revoked as well, and the user is deleted from the catalog table SYSUSERAUTH. Revoking CONNECT authority does *not* cause objects owned by that user to be dropped; if they should be dropped, this can be done by a user with DBA authority.

Revoking DBA authority automatically causes all other authorities except CONNECT to be revoked. Revoking RESOURCE or SCHEDULE authority implies no other revocations.

Defining Privileges

The system keeps track of the *privileges* that each authorization ID has, and makes sure that each ID performs only authorized operations on the database.

Authorized users can create and drop tables or views, and compile and run programs that operate on these tables or views. Anyone who creates a table or view or compiles a program can selectively share the use of that table, view, or program with other authorization IDs.

The privileges you need vary depending on what operations you want to perform. There are two categories of privileges: privileges on tables and views, and privileges on programs.

Defining Privileges on Tables and Views

You can have any or all of the following privileges on specific tables and views:

ALTER	Privilege to add new columns and keys to a table (does not apply to views)
DELETE	Privilege to delete rows from tables and views
INDEX	Privilege to create new indexes on a table (does not apply to views)
INSERT	Privilege to insert new rows into tables or views
REFERENCES	Privilege to add, drop, activate, or deactivate a foreign key relationship (does not apply to views)
SELECT	Privilege to retrieve data from tables or views
UPDATE	Privilege to change column values in tables or views.

When you create a new table or view, you are automatically given full privileges on it. In most situations, you are also given the GRANT option on each privilege which enables you to grant any or all of these individual privileges to other authorization IDs. When you grant a privilege, you may include the GRANT option so that the recipient will be able to grant the privilege to others in turn.

If you grant the privileges on an object to PUBLIC, all authorization IDs (including those that do not yet exist) will have the same privileges that you have.

If you have DBA authority, you have the same privileges on an object and you can grant those privileges (or drop the object) in the same way that the owner of the object can.

Any privilege that you hold on a table or view may be exercised directly through ISQL and the DBS utility as well as application programs.

Privileges on tables and views are listed in the database manager catalog tables. SYSTABAUTH and SYSCOLAUTH. To check what privileges you hold or have granted to other authorization IDs, make the suitable queries on these tables. See the *DB2 Server for VSE & VM SQL Reference* manual for more information on the catalog tables.

Revoking Privileges

Once you have granted a privilege, you can revoke it by issuing a REVOKE statement. (You can never revoke a privilege from yourself.) If you revoke a privilege from user LEENA, it is automatically revoked from all authorization IDs to whom LEENA granted it, unless the other authorization IDs have another independent source for the same privilege. The most common and most convenient way to enter a REVOKE statement is through ISQL or the DBS utility. You can code REVOKE statements within a program; however, because the user ID and passwords in the REVOKE statements cannot be host variables, the statements have limited use.

If you attempt to revoke a privilege that is currently in use by a running program, the REVOKE statement is queued until the program ends its current *logical unit of work*. For example, if you revoke the UPDATE privilege from user MARY, but MARY's program is running and is already making updates, your REVOKE statement does not take effect until MARY's updates are finished.

The database manager can also automatically revoke privileges on views, or drop the view definition. Suppose BILL grants GENE the SELECT privilege with the GRANT option on the EMPLOYEES table. GENE then defines a view called SALARY on this table, and grants the SELECT privilege on that view to other users. After some time, BILL decides to revoke the SELECT privilege on the EMPLOYEES table from GENE. When BILL does so, the system also automatically revokes the SELECT privilege from SALARY also, including all SELECT privileges on SALARY that GENE passed on. If after this process GENE holds no privileges on SALARY, the definition of SALARY is dropped.

Defining Privileges on Packages

Assigning User Privileges to the Owner

Application programs must be preprocessed before they are compiled or assembled. Successfully preprocessing an application program results in the creation or replacement of a package in the database. The contents of the package are then used to satisfy database requests at run time.

When the package is created, the system determines the level of the RUN privilege to be given to the owner (EXECUTE privilege can be used as a synonym for RUN privilege). This depends on such factors as the preprocessed SQL statements, the existence and ownership of the referenced objects (tables, indexes, dbspaces, and so on), and the owner's authorization level (DBA, RESOURCE, SCHEDULE, or CONNECT).

The owner of a package is assigned the RUN privilege based on the following rules:

- If the owner does *not* have DBA authority, the RUN privilege is assigned when the preprocessor successfully creates or replaces the package.
- If the owner has DBA authority, the RUN privilege is assigned when none of the preprocessed SQL statements depends on the owner having DBA authority.

There is an exception to this rule: if an SQL statement selects information from a table on which the owner does not have the explicit SELECT privilege, and the owner has DBA authority, then the owner may still be assigned the RUN privilege. This will depend on the result of preprocessing all the other SQL statements in the program.

When a particular SQL statement references objects that do not exist or have different attributes at preprocessing time, the system still creates a package for the program and assigns RUN privilege to the owner. In this case, the required objects must be correctly defined at run time, or execution of the program will fail.

In fact, the determination of whether an owner receives the RUN privilege is based on the aggregate “*score*” of all preprocessed SQL statements in the program. Each statement is individually assigned an authorization score; at the end of the preprocessing phase, the system picks the lowest score, and assigns that to the owner.

The scores, and the decision tables used to assign them, are discussed in Appendix F, “Decision Tables to Grant Privileges on Packages” on page 339.

Assigning Privileges to Others

The database manager provides a GRANT statement that allows the owner of a package to grant the RUN privilege on the package to other users.

Determining When the Owner Can Grant the RUN Privilege: The owner of a package is assigned the GRANT RUN privilege when all preprocessed SQL statements in the program allow the owner to GRANT RUN. If the owner can grant the RUN privilege on a package, a user with DBA authority has the same ability.

Circumstances which enable an owner to gain the GRANT RUN privilege include:

- The owner has the necessary privileges (with the GRANT option) to access any referenced objects.
- The package does not contain any statements that require DBA authority. The following are examples of operations that require DBA authority:
 - Acquiring a public dbspace
 - Creating a table in another user's dbspace or in a SYSTEM dbspace
 - Acquiring a dbspace for another user
 - Altering another user's table when the owner doesn't have explicit ALTER authority on the table
 - Locking another user's dbspace
 - Commenting on another user's table
 - Dropping another user's object
 - Locking another user's table
 - Altering another user's dbspace
 - Creating an index on another user's table when the owner doesn't have explicit INDEX authority on that table
 - Creating a table for another user
 - Inserting, deleting, or updating another user's table when the owner doesn't have the explicit authority to do so.

Note: The following statements also require DBA authority, but do not affect the RUN privilege, because they are not checked until run time (when they may be rejected).

- ALTER DBSPACE when the owner qualifier is not given
- LOCK DBSPACE when the owner qualifier is not given
- DROP DBSPACE when the owner qualifier is not given
- CREATE TABLE in someone else's dbspace or in a SYSTEM dbspace when the DBSPACE owner qualifier is not given.

Differences Between Static and Dynamic Statements

There is a difference between static, dynamic, and extended dynamic SQL statements, when determining the privileges of the owner and other users of the package being run.

Static

At preprocessing time the objects referenced in static statements are checked for existence, for usage consistent with the definitions in the database, and to determine whether the package owner has the required privileges. This process allows the person who is preprocessing a package to encapsulate a set of object privileges that he or she possesses into that package and to subsequently grant them to others.

Dynamic

All dynamic statements are checked at the time the PREPARE or EXECUTE IMMEDIATE statement is run and the privileges on the objects referenced in the statement are checked against those of the authorization ID of the runner of the package. There is, therefore, no way to encapsulate object privileges with dynamic statements.

Extended Dynamic

For modifiable packages, all statements are checked against the privileges of the person who is preparing or modifying the package, as per static SQL. For nonmodifiable packages, statements prepared with extended PREPARE Filling Empty Section statement are checked as per dynamic SQL, and statements prepared with the other forms of extended PREPARE are checked as per static SQL.

Revoking the Run Privilege

The REVOKE statement may be used to revoke the RUN privilege on a package in the same way it revokes privileges on tables and views.

In some situations, the system automatically revokes the RUN privilege from a number of users. Suppose user GENE has preprocessed a program that makes use of some privilege, such as SELECT. GENE receives the RUN privilege on the package with the GRANT option, and grants this privilege to other users.

If the SELECT privilege is now revoked from GENE, the package associated with the program is automatically marked invalid. When the program is run (by GENE or any other user), the system attempts to regenerate a valid (fully authorized)

package. At the time of this regeneration process, the following outcomes are possible:

1. GENE has all the privileges required by the program, and furthermore has the GRANT option on all these privileges. In this case, the package is regenerated, all existing grants of the RUN privilege on the program remain in effect, and execution proceeds normally.
2. For some SQL statements in the program, GENE lacks the necessary privilege, or has the privilege without the GRANT option. In this case, GENE retains the RUN privilege on the program, but all existing grants of the RUN privilege are revoked. When the program is run, those SQL statements for which GENE has the necessary privilege execute successfully, and others return error codes.

Recording Assigned Privileges in the Catalog Tables

The database manager records the current RUN and GRANT RUN privileges held by all authorization IDs in the SYSPROGAUTH catalog table. The entries in the catalog identify:

- The grantor
- The grantee
- The package that is the subject of the RUN privilege
- A marker indicating that the grantee holds either RUN ('Y') or GRANT RUN ('G') authority.

The entries are added to the catalog tables as an application is preprocessed. The entries may depend, of course, on whether the package satisfies the various conditions described in the preceding sections. The system also makes entries in the SYSPROGAUTH catalog table when someone grants the RUN privilege to another authorization ID.

The system also updates the SYSUSERAUTH, SYSCOLAUTH, and SYSTABAUTH catalog tables. The package's dependency on some authorization is recorded in these catalog tables. For example, when a package requires RESOURCE authority to execute successfully, an entry is made in SYSUSERAUTH to reflect that dependency. The system uses the catalog table entries to keep track of valid and invalid packages.

Chapter 10. Special Topics

Contents

Using Datetime Values with Durations	222
Using Durations	222
Resolving Peculiarities of Date Arithmetic	222
Summarizing Addition Operations	224
Summarizing Subtraction Operations	225
Using Field Procedures	225
Assigning Field Procedures to Columns	227
Understanding Field Procedure Rules	227
Input from an Application Program	228
Output to an Application Program	228
Comparison	228
Referential Integrity	228
Scalar Functions	229
Column Functions	230
Concatenation	230
The IN and BETWEEN Predicates	230
The LIKE Predicate	230
Sorting	230
Null Values	230
Unions and Joins	230
Sub-SELECTS	231
Ensuring Data Integrity	231
Ensuring Entity Integrity	231
Using Unique Constraints	232
When Creating a View	232
Ensuring Referential Integrity	232
Defining Terms	232
Ensuring Referential Integrity in New Tables	234
Adding Referential Integrity to Existing Tables	235
Managing Table Relationships	236
Modifying Applications to Ensure Integrity	237
Modifying Data in Tables Containing Referential Constraints	237
Restrictions on Using Delete Rules	240
Generating SQL Statements in Response to Table Modifications	241
Enforcing Referential Integrity	242
Removing Referential Constraints	244
Switching Application Servers	244
Identifying Switching Options	245
How to Switch Servers	245
Accessing a New Application Server	245
Illustrating Sample Code	246
Preprocessing the Program on Multiple Application Servers	247
Condition Handling with LE/VSE	247

Using Datetime Values with Durations

Using Durations

A duration is a value that represents an interval of time. The value may be a constant, a column name, a host variable, a function, an expression, or an expression followed by a duration attribute. Numbers are interpreted as durations only in certain contexts as defined in the *DB2 Server for VSE & VM SQL Reference* manual; the arithmetic of using date, time, and timestamp is discussed in detail. Figure 78 on page 224 and Figure 79 on page 225 summarize this topic.

Resolving Peculiarities of Date Arithmetic

What does it mean to add a month to a given date? Presumably the result should be the same day of the next month. That is, one month after January 1 is February 1, and one month after February 1 is March 1. But what is one month after January 31? This difficulty (which is the reason why certain contracts are always dated the first of the month) is resolved by the further assumption that the result should be the last day of February. Thus, adding a month to a given date gives the same day of the next month *except when the next month does not have such a day*, in which case the result is the last day of that month. But, one month from the last day of a month is not necessarily the last day of the next month. One month from the last day of February, for example, is not the last day of March. Thus (a date) + (a simple duration of months) - (a simple-duration of months) is not necessarily equal to the original date.

The definition of a month does not permit a consistent system of date arithmetic. If this is a problem, it can be avoided by using days rather than months. For example, to increment the date *date3* by the difference between the dates *date1* and *date2*, the expression:

```
DATE ( DAYS(date1) - DAYS(date2) + DAYS(date3) )
```

will give an accurate result whereas *date1* - *date2* + *date3* may not. Figure 77 on page 223 shows how SQLWARN7 provides warnings during date arithmetic when the resulting date has to be adjusted to derive a valid date.


```

Let D1 be the DATE 1994-02-29, a leap year:
                                                    SQLWARN7

D1 + 1 DAY      = 1994-03-01                ' '
D1 + 2 MONTHS  = 1994-04-29                ' '
D1 + 1 YEAR    = 1995-02-28                'W'
D1 + 4 YEARS   = 1998-02-29                ' '

Let N be DEC(8,0) and set to 00010203.
D1 + N
      = 1994-02-29 + 1 YEAR + 2 MONTHS + 3 DAYS
      = 1995-02-28 + 2 MONTHS + 3 DAYS      'W'
      = 1995-04-28 + 3 DAYS
      = 1995-05-01

Let D2 be the DATE 1995-03-31:
                                                    SQLWARN7

D2 + 1 MONTH   = 1995-04-30                'W'
D2 + 2 MONTHS = 1995-05-31                ' '

```

Figure 77. Setting SQLWARN7 during Date Arithmetic. When incrementing or decrementing dates, SQLWARN7 is set when the resulting date is an invalid date because of a leap year or month difference, and a valid date is derived.

Summarizing Addition Operations

DATETIME ADDITION = OPERAND + OPERAND

LEFT OR RIGHT OPERAND ↓	LEFT OR RIGHT OPERAND													RESULT DATA TYPE ↓	
	DURATIONS														
					SIMPLE										
	DATE	TIME	TIME STAMP	STRING	DATE	TIME	TIME STAMP	YEAR	MONTH	DAY	HOURL	MINUTE	SECOND		MSECONDS
DATE					X			X	X	X					DATE
TIME						X					X	X	X		TIME
TIME STAMP					X	X	X	X	X	X	X	X	X	X	TIME STAMP

Figure 78. Datetime Addition

- An X denotes valid datetime addition operation.
- STRING means a character string in a valid datetime format.

Summarizing Subtraction Operations

DATETIME SUBTRACTION = MINUEND - SUBTRAHEND

MINUEND ↓	SUBTRAHEND														RESULT DATA TYPE ↓	
	DURATIONS															
					SIMPLE											
	DATE	TIME	TIME STAMP	STRING	DATE	TIME	TIME STAMP	YEAR	MONTH	DAY	HOUR	MINUTE	SECOND	MILLISECOND		
DATE	1			1	2			2	2	2						1=(8,0) 2=DATE
TIME		1		1		2					2	2	2			1=(6,0) 2=TIME
TIME STAMP				1	1	2	2	2	2	2	2	2	2	2		1=(20,6) 2=TIME STAMP

Figure 79. Datetime Subtraction

- 1 or 2 denotes a valid datetime subtraction operation.
- 1 means a result data type of DECIMAL(8,0), DECIMAL(6,0) or DECIMAL(20,6) which is deemed as a date duration, time duration, or timestamp duration respectively. 2 means a result data type of date, time, or timestamp.
- STRING means a character string in a valid datetime format.

Using Field Procedures

Field procedures enable you to alter the sorting sequence of values entered in a single short string column (CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC). For some applications the standard EBCDIC sorting sequence is not appropriate. For example, telephone directories sometimes require that names like “McCabe” and “MacCabe” appear next to each other, and the standard sorting routine would separate them. Another example is a national language character set that does not use the Roman alphabet. For example, Kanji (Japanese) can only be sorted properly using a field procedure.

If you assign a field procedure to a column, it is called whenever values in that column are changed or are inserted, and it transforms (encodes) the original value into one value that sorts properly.

When you retrieve a row from the encoded column, the same field procedure decodes it into the original form. You will never see the encoded string. From a user's point of view, all a field procedure does is change the sorting sequence for a column.

For example, consider a table with a short string column that contains the four divisions in a company: North, South, East, and West. Divisions are usually sorted as follows:

```
East
North
South
West
```

You can, however, write a field procedure that encodes North as 1, South as 2, East as 3 and West as 4. The divisions would then be sorted as follows:

```
North
South
East
West
```

Note: The encoded values do not have to be the same data type as the decoded values. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for a description of the catalog table SYSCOLUMNS, which contains the descriptions of decoded columns, and SYSFIELDS, which contains the descriptions of the corresponding encoded columns.

While field procedures are used primarily to alter the standard EBCDIC sorting sequence, they can also be used in any application program that requires short strings to be stored differently from how they are inserted or retrieved.

For a sample field procedure and the rules for writing field procedures, refer to the *DB2 Server for VSE System Administration* manual.

DB2 Server for VSE provides two field procedures for performing cultural sorts. They are:

FP870L2 Sample field procedure for cultural sorting for the Latin 2 code page (Regions: Slovenia, Poland, and Romania).

FP102CY Sample field procedure for cultural sorting for the Cyrillic code page (Regions: Russia, Bulgaria, Serbia, and Montenegro).

| If Data Propagator Capture for VSE is being used on tables that have columns with
| field procedures, "1-way" field procedures must be defined on the Data Propagator
| Change Data (CD) tables to properly propagate this data. Refer to the *DB2 Server
| for VSE System Administration* manual for more information.

Assigning Field Procedures to Columns

To assign a field procedure to a new column, include the FIELDPROC clause on either the CREATE TABLE or ALTER TABLE statement. To assign field procedures to columns in an existing table, you must unload the data, recreate the table to include the field procedures, and they reload the data back into the table. If you create a column without a field procedure, you cannot add one later.

Refer to the *DB2 Server for VSE & VM SQL Reference* manual for the syntax diagrams for the CREATE and ALTER TABLE statements. The fieldproc-block for these diagrams is shown below.

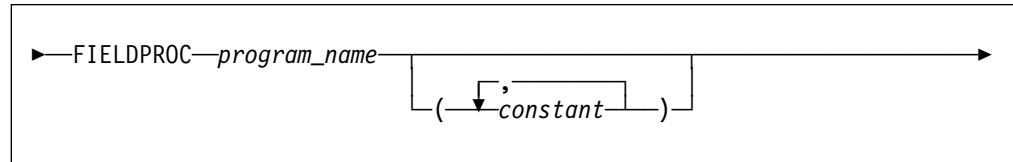


Figure 80. fieldproc-block Syntax

For example:

```
ALTER TABLE SCOTT.SUPPLIERS ADD RATING CHAR(6) FIELDPROC MYFLDPRO (10,5)
```

The constants **(10,5)**, that follow the *program_name* **MYFLDPRO** are optional parameters, defined when the field procedure is written and passed to the field procedure when it is invoked.

Understanding Field Procedure Rules

In most cases you will not have to worry about the rules that define when a field procedure encodes or decodes a short string. However, if you understand when the database manager calls field procedures, this can help you understand their performance implications. The less you call field procedures to encode or decode strings the better your application's performance will be.

Understanding when field procedures are called can also help you to avoid some pitfalls. For example, consider a table TABLE_A with a column COLUMN_A that has fieldproc F1, and consider these two statements:

```
SELECT SUBSTR(MAX(COLUMN_A,1,5)) FROM TABLE_A
```

```
SELECT MAX(SUBSTR(COLUMN_A,1,5)) FROM TABLE_A
```

You might assume that the two statements should return essentially the same result; however, different results can be returned depending on your coding. In the first statement, the database manager does the following:

1. Finds the maximum encoded value in COLUMN_A
2. Decodes the result from MAX with field procedure F1
3. Applies the SUBSTR function to the decoded value of the result from MAX.

In the second statement, the database manager does the following:

1. Decodes the value in COLUMN_A with field procedure F1
2. Applies the SUBSTR function to the decoded value in COLUMN_A
3. Applies the MAX function to the result of the SUBSTR function.

That is, the first statement MAX is applied to encoded values, and the second is applied to decoded values.

The rest of this section covers the rules that define when a field procedure encodes or decodes a short string.

Input from an Application Program

The field procedure is called to encode data when your application program inserts or updates data. This includes the following statements:

- INSERT
- PUT
- UPDATE

Output to an Application Program

The field procedure is called to decode data when your application program fetches or selects data. This includes the following statements:

- FETCH
- SELECT INTO

Comparison

If a column with a field procedure is compared to a constant, the constant is first encoded by the field procedure. The comparison is then performed between the encoded values in the column and the encoded value of the constant.

Host-variables, parameter markers, and the USER special register are treated the same way.

For example, consider the following SQL statement where COLUMN_A has field procedure F1:

```
SELECT * FROM MY_TABLE WHERE COLUMN_A > 'SMITH'
```

When processing the above statement, the database manager first encodes 'SMITH', and then for each row in MY_TABLE, compares F1 to the encoded value in COLUMN_A.

A field procedure can only encode short strings values. If the variable or constant is of a data type other than CHAR, VARCHAR, GRAPHIC or VARGRAPHIC, a negative SQLCODE is returned.

If a column with a field procedure is compared to another column, both columns must have field procedures with the same *program_name*, comparable encoded data type, and the same CCSID. If not, a negative SQLCODE is returned.

Referential Integrity

If a primary key column has a field procedure, then the foreign key column must have the same field procedure, and the CCSIDs of both key columns must be the same. Otherwise, a negative SQLCODE is returned. For two field procedures to be the same, their *program_names*, encoded data type, encoded data length, and input parameters must be identical.

For example, the following is correct:

```

CREATE TABLE PRIMARY
  (COLUMN_A CHAR(10) FIELDPROC F1 NOT NULL,
   COLUMN_B INTEGER)
  PRIMARY KEY(COLUMN_A)

CREATE TABLE FOREIGN
  (COLUMN_A CHAR(10) FIELDPROC F1 NOT NULL,
   COLUMN_B CHAR(10))

ALTER TABLE FOREIGN
  ADD FOREIGN KEY (COLUMN_A)
  REFERENCES PRIMARY ON DELETE SET NULL

```

Scalar Functions

All scalar functions operate on decoded values. For example, if 'V' is a string in a column with a field procedure, HEX('V') returns the hexadecimal representation of 'V'. The result is not associated with the original column's field procedure. However, if the result of a scalar function is compared to a column that is associated with a field procedure, this result is encoded by the comparison column's field procedure. The comparison is then made between the encoded value of the column and the encoded result of the scalar function. This is consistent with how columns with field procedures are compared to constants.

For example:

1. Consider a table (MY_TABLE) with COLUMN_A that has field procedure F1 and COLUMN_B that has field procedure F2. Consider the following SQL statement:

```
SELECT * FROM MY_TABLE WHERE COLUMN_A > SUBSTR(COLUMN_B,3,3)
```

For each row of MY_TABLE, the following occurs:

- a. The encoded values of COLUMN_B are decoded by field procedure F2.
 - b. The substring operation is applied to the decoded value of COLUMN_B.
 - c. The result of the substring operation is encoded by field procedure F1.
 - d. Finally, the encoded value of COLUMN_A is compared to the encoded result of the substring operation.
2. Consider a table (MY_TABLE) with three columns, where COLUMN_A has field procedure F1, COLUMN_B has field procedure F2, and COLUMN_C is NOT NULL and has field procedure F3. Consider the following SQL statement:

```
SELECT * FROM MY_TABLE WHERE COLUMN_A > VALUE(COLUMN_B,COLUMN_C)
```

For each row of MY_TABLE, the following occurs:

- a. If the value of COLUMN_B is not null, then COLUMN_B is decoded, using F2. Call the result 'M'.
- b. If the value of B is null, then C is decoded, using F3. Call the result 'M'.
- c. 'M' is then encoded using F1.
- d. The encoded result of the VALUE function is then compared to the encoded value of COLUMN_A.

Note: A field procedure is never called to encode or decode a NULL value. A NULL value always maps to a NULL.

3. If a column with a field procedure is the argument of the LENGTH function, first it is decoded by the field procedure, and then the length of the result is returned. Of course, if the column data type is a fixed length (for example,

CHAR(15)), there is no need to actually decode the column value. The length returned by the function is simply the fixed length of the column (15 in this example).

Column Functions

The column functions MAX and MIN operate on encoded values. The remaining column functions operate on numeric data, and are not affected by field procedures.

Concatenation

The concatenation operator is basically a scalar function, and follows the same rules as a scalar function.

For example, consider a table (MY_TABLE) with COLUMN_A that has field procedure F1 and COLUMN_B that has field procedure F2. Now, consider the following SQL statement:

```
SELECT * FROM MY_TABLE WHERE COLUMN_A > 'ADDITION' CONCAT COLUMN_B
```

For each row in MY_TABLE, the following occurs:

1. The value of COLUMN_B is decoded by F2.
2. 'ADDITION' is concatenated with the decoded value in COLUMN_B.
3. The result of the concatenation is encoded by field procedure F1.
4. The encoded result of the concatenation is compared to the encoded value of COLUMN_A.

The IN and BETWEEN Predicates

These predicates operate the same as a comparison between a column with a field procedure and a constant.

The LIKE Predicate

This predicate operates on decoded values.

Sorting

Indexes will be based on encoded values. The ORDER BY and GROUP BY clauses will sort the data according to the encoded format. The database manager also sorts values during a UNION operation.

Null Values

While a column with a field procedure may be defined to allow null values, the field procedure is never called to process a null value. A decoded null value always maps to an encoded null value, and an encoded null always maps to a decoded null.

Unions and Joins

The rules for comparing two columns with field procedures apply to unions and joins. The two columns must have the same field procedure.

Sub-SELECTS

All the rules described above apply to sub-SELECTs.

For example:

```
SELECT * FROM TABLE_1
      WHERE COLUMN_A=(SELECT COLUMN_B FROM TABLE_2);
```

```
SELECT * FROM TABLE_1
      WHERE COLUMN_A IN (SELECT COLUMN_B FROM TABLE_2);
```

If the columns COLUMN_A and COLUMN_B have different field procedures these statements are invalid (field procedure comparison rules apply). For example:

```
INSERT INTO T1 (COLUMN_A) SELECT COLUMN_B FROM T2;
```

In this statement, the decoded data types for COLUMN_A and COLUMN_B must be compatible. If so, the value in COLUMN_B will be decoded with F2. The decoded value is then encoded by F1, and the resulting value is inserted into COLUMN_A.

Ensuring Data Integrity

Data integrity refers to the accuracy and correctness of data in the database. When related changes are made to a database, the database manager maintains integrity of the data by ensuring that either all or none of the changes are made. This protects other users and programs from using inconsistent or wrong data. This type of integrity is called *atomic integrity*.

Data integrity is also maintained by ensuring the uniqueness of certain data in the database. For example, the SUPPLIERS table must not have duplicate supplier numbers (SUPPNO). Using this integrity rule, the database manager ensures that duplicates do not exist. This type of integrity is called *entity integrity*.

For consistency and integrity, when one table references values in another table, the referenced values must exist in both tables, or the reference is not valid. The database manager automatically enforces rules that you define on the tables. These rules are called *referential constraints*. Enforcement of referential constraints ensures the *referential integrity* of the data referenced.

Ensuring Entity Integrity

The rule that each row in the EMPLOYEE table must represent one and only one employee is an example of entity integrity. By defining a *primary key* on the table, you can ensure that duplicate rows do not occur, thereby enforcing entity integrity. For example, in the following SQL statement, the column EMPNO is defined as a primary key, so a unique index is automatically created on that column. This enforces uniqueness of the data in that column.

```
CREATE TABLE EMPLOYEE
(EMPNO      CHAR(6)      NOT NULL,
 FIRSTNAME  VARCHAR(12)  NOT NULL,
 LASTNAME   VARCHAR(15)  NOT NULL,
 SALARY     DECIMAL(9,2) ,
 PRIMARY KEY (EMPNO)
)
```

Using Unique Constraints

A unique constraint enables you to enforce data integrity without having to enforce entity integrity. While a primary key can ensure that each row in the EMPLOYEE table represents one and only one employee, a unique constraint can ensure that each entry in another column is unique. For example, a company has one telephone for every employee and wants to maintain a set of unique phone numbers. Its database, however, already uses an employee number as a primary key. A unique constraint can ensure that no phone numbers are repeated in the table. Also, if the phone number consists of several columns (area code, 7-digit number, extension), the unique constraint can include all those columns.

```
CREATE TABLE EMPLOYEE
(EMPNO      CHAR(6)      NOT NULL,
 FIRSTNME  VARCHAR(12)  NOT NULL,
 LASTNAME  VARCHAR(15)  NOT NULL,
 AREACODE  CHAR(3)      NOT NULL,
 PHONENUM  CHAR(7)      NOT NULL,
 PHONEEXT  CHAR(4)      NOT NULL,
 PRIMARY KEY (EMPNO)
 UNIQUE PHONE (AREACODE,PHONENUM,PHONEEXT)
)
```

The ALTER TABLE command can be used to add, activate, deactivate, or remove a unique constraint. Another way to remove a unique constraint is either by dropping the table or the dbspace. Although a unique index is created when the unique constraint is created, the constraint cannot be dropped by dropping the index.

When Creating a View

The WITH CHECK OPTION clause in the CREATE VIEW statement is an example of data integrity in the maintenance of data defined by a view. See “Creating a View” on page 62.

Ensuring Referential Integrity

Defining Terms

Referential integrity defines the condition on a set of tables in which the existence of values in one table depends on the existence of the same values in another table. By enforcing referential constraints (referential integrity rules) that are part of the table definitions, the database manager ensures the referential integrity of the data in the tables.

Figure 81 on page 233 shows examples of relationships supported by the database manager.

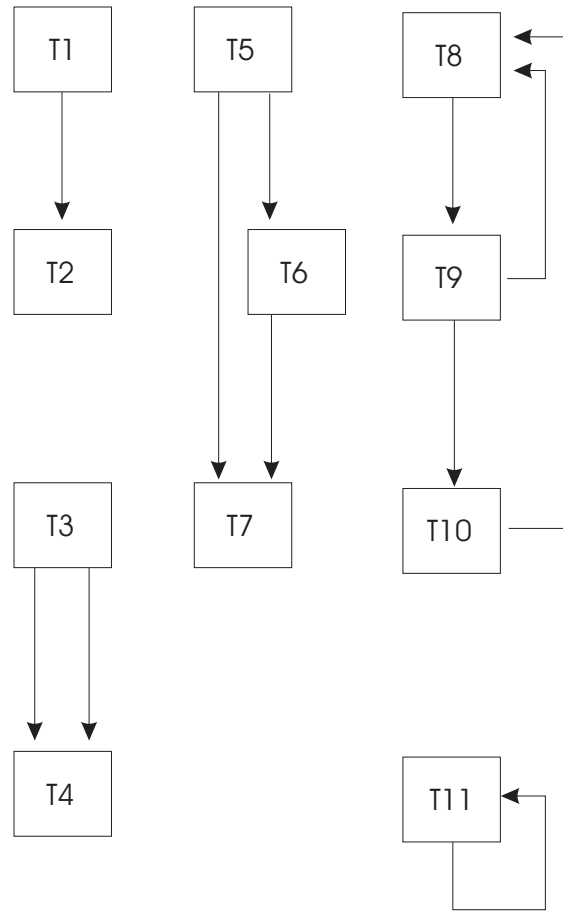


Figure 81. Table Relationships with Referential Integrity. T1, T2, ... are tables. Arrows point from parent tables to dependent tables.

You should be familiar with the following terms:

Relationship

A relationship is formed by connecting two tables directly. The tables are related through matching column values in the tables. For example, in Figure 81, tables T1 and T2 show a simple relationship. T3 has two relationships with T4. T5 has two *paths* to T7 (one directly, the other through T6), but only one *relationship* with T7. T5 also has a relationship with T6. Tables are connected to each other when relationships are formed.

Referential Constraint

A relationship between a primary key and a foreign key, along with a set of rules that define how the relationship is maintained. This relationship is that every foreign key value must match a primary key value or be null.

Referential Cycle

A set of referential constraints such that each table in the set is a descendent of itself.

Referential Structure

A set of tables that are related to each other by referential constraints. For example, T5 is a parent of both T6 and T7, which are its dependents. T7 is also a dependent of T6.

Parent Table	A table whose primary key is referenced in a referential constraint. For example, T1 is the parent of T2.
Dependent Table	A table with a foreign key that is related to another table (the parent) through a referential constraint. For example, T4 is a dependent of T3.
Delete-Connected Table	A table that may be involved in a delete operation on another table.
Descendent Table	A table is a descendent table if it is a dependent table or a dependent of a descendent table. For example, in Figure 81 on page 233, both T6 and T7 are descendent tables of T5.
Parent Row	A row in a parent table with a primary key value that is referenced by the foreign key value in at least one row in a dependent table.
Dependent Row	A row in a dependent table with a foreign key value that matches a primary key value in the parent table referenced in the referential constraint.
Self-Referencing Table	A self-referencing table is both the parent and the dependent table in the same relationship. This relationship is not supported by the DB2 Server for VSE product. For example, T11 is a self-referencing table.
Primary Key	A set of non-null columns that together uniquely identify every row in a table. The values in these columns are known as <i>primary key values</i> .
Foreign Key	A set of columns whose values are called <i>foreign key values</i> . A foreign key only exists as part of a referential constraint.

Ensuring Referential Integrity in New Tables

To ensure referential integrity in new tables, you must specify a primary key, a foreign key, and a delete rule that together define the relationship between the parent table and the dependent table. Delete rules specify what will happen to the dependent rows if the corresponding parent row is deleted. Insert and update rules are automatically defined on tables when primary keys and foreign keys are defined on those tables.

The relationship is defined when the new table is created using the CREATE TABLE statement.

You should be aware of the referential constraints of the tables you manipulate, as well as the rules for those tables. In this way you can avoid violating any referential constraints, and take appropriate action should you inadvertently do so.

In the example below, the EMPLOYEE table is the parent of the DEPARTMENT table. This relationship is established by specifying a primary key (EMPNO) on the EMPLOYEE table and a foreign key (MGRNO) on the DEPARTMENT table. This relationship specifies that every manager listed in the DEPARTMENT table is also

listed in the EMPLOYEE table. The REFERENCES privilege is required on the parent table. The foreign key is nullable.

```

CREATE TABLE EMPLOYEE
  (EMPNO      CHAR(6)      NOT NULL      ← primary key
   FIRSNME    VARCHAR(12)   NOT NULL
   MIDINIT    CHAR(1)      NOT NULL
   LASTNAME   VARCHAR(15)  NOT NULL
   WORKDEPT   CHAR(3)      ,
   PHONENO    CHAR(4)      ,
   SALARY     DECIMAL(9,2) ,
   PRIMARY KEY (EMPNO)    )

CREATE TABLE DEPARTMENT
  (DEPTNO     CHAR(3)      NOT NULL      ← primary key
   DEPTNAME   VARCHAR(36)  NOT NULL
   MGRNO      CHAR(6)      , ← foreign key
   PRIMARY KEY (DEPTNO)
   FOREIGN KEY MNUM (MGRNO)
     REFERENCES EMPLOYEE ON DELETE SET NULL)

```

Adding Referential Integrity to Existing Tables

To add referential integrity to existing tables, you must add a primary key, a foreign key, and a delete rule that together define the relationship between the parent table and the dependent table. Delete rules specify what will happen to the dependent rows if the corresponding parent row is deleted. Insert and update rules are implicitly defined on tables when primary keys and foreign keys are defined on those tables.

The relationship is defined using the ALTER TABLE statement.

When keys (primary or foreign) are added to an existing table, any packages that depend on the table are invalidated. When the application programs are run again, the packages will be dynamically reprocessed. Refer to “Running the Program” on page 134 for more information on dynamic reprocessing.

As in the case of new tables, you should be aware of the referential constraints of the tables you manipulate as well as the rules for those tables, in order to avoid violating any referential constraints or to take appropriate action should you inadvertently do so.

Consider the existing DEPARTMENT and PROJECT tables. The PROJECT table was created by the following CREATE TABLE statement:

```

CREATE TABLE PROJECT
  (PROJNO     CHAR(6)      NOT NULL      ← primary key
   PROJNAME   VARCHAR(24)  NOT NULL
   DEPTNO     CHAR(3)      NOT NULL      ← foreign key (To be added)
   RESPEMP    CHAR(6)      NOT NULL
   PRSTAFF    DECIMAL(5,2) ,
   PRIMARY KEY (PROJNO)    )

```

The following ALTER TABLE statement adds a referential constraint to the PROJECT table, thereby establishing a relationship between it and the existing DEPARTMENT table:

```
ALTER TABLE PROJECT
  ADD FOREIGN KEY DNUM (DEPTNO)
  REFERENCES DEPARTMENT ON DELETE CASCADE;
```

In this relationship, DEPARTMENT is the parent table and PROJECT is the dependent table. This specifies that every department that is responsible for a project is also in the DEPARTMENT table.

Note: The ALTER TABLE statement can also be used to defer the enforcement of referential constraints or cause the removal of referential constraints. These topics are discussed in the section “Enforcing Referential Integrity” on page 242.

Managing Table Relationships

The ALTER TABLE statement can be used to add, drop, activate, or deactivate primary and foreign keys. Various clauses of the statement alter alter the keys and to establish relationships between tables. When the ALTER TABLE statement establishes or changes relationships, specific privileges are required on parent tables and dependent tables. Figure 82 shows the privileges that are required.

Figure 82. Privileges to Use the ALTER TABLE Statement

ALTER TABLE Clause	Privilege on Parent Table	Privilege on Dependent Table
Add Column	ALTER	
Add Primary Key	ALTER	
Add Foreign Key	REFERENCES	ALTER
Drop Primary Key	ALTER REFERENCES ¹	ALTER
Drop Foreign Key	REFERENCES	ALTER
Deactivate Primary Key	ALTER REFERENCES ¹	ALTER
Deactivate Foreign Key	REFERENCES	ALTER
Activate Primary Key	ALTER REFERENCES ¹	ALTER
Activate Foreign Key	REFERENCES	ALTER

Note: The REFERENCES privilege is required only if the parent table has dependents.

You can grant to or revoke from another user the privilege to add, drop, activate, or deactivate a relationship between a parent table and its dependent. In order to enter any of these statements, you must have the REFERENCES privilege on the parent table whenever a referential constraint is to be:

- Created on a new table (CREATE TABLE)
- Added to an existing table (ALTER TABLE)

- Dropped, activated, or deactivated (ALTER TABLE).

By revoking the privileges previously granted on tables in a referential structure, you can prevent the accidental removal of constraints that your applications may depend on.

Modifying Applications to Ensure Integrity

Applications that currently enforce consistency and integrity of their data can be modified to let the database manager do the checking. Using the referential constraints and the integrity rules that apply to the tables containing the data, the system checks that the rules are adhered to, and thereby enforces integrity of the data. As this function can be performed by the database manager, some existing code can be removed from the application.

Modifying Data in Tables Containing Referential Constraints

To maintain the consistency and integrity of the data, the database manager checks that integrity rules for insert, update, and delete operations are followed.

Applying Insert Rules: The database manager checks the implicit insert rules when a row is inserted into either the parent or a dependent table in a referential structure. When a row is inserted into a parent table, the database manager checks that the primary key remains unique and does not contain null values. When a row is inserted into a dependent table, the database manager checks each foreign key for the following:

- Each has a matching primary key in the parent table, or
- Each contains a null value in one or more of its columns.

Assuming for the moment that department D21 does not already exist in the parent table (DEPARTMENT), the following INSERT statement adds a new row to DEPARTMENT.

```
INSERT INTO DEPARTMENT (DEPTNO,DEPTNAME,MGRNO,ADMRDEPT)
VALUES ('D21','ADMINISTRATION SYSTEMS','000070','D01')
```

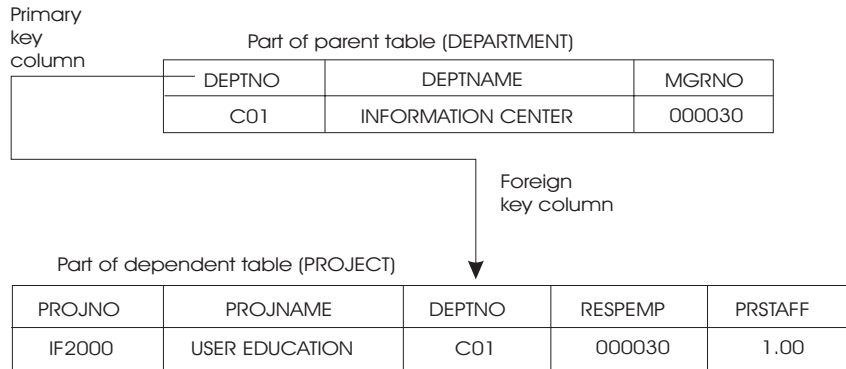
Note: The primary key (the DEPTNO column) in the DEPARTMENT table remains unique and does not contain null values.

Figure 83. Part of Department Table

DEPTNO	DEPTNAME	MGRNO
A00	SPIFFY COMPUTER SERVICE DIV.	000010
B01	PLANNING	000020
C01	INFORMATION CENTER	000030
D01	DEVELOPMENT CENTER	?
D11	MANUFACTURING SYSTEMS	000060
E11	OPERATIONS	000090
D21	ADMINISTRATION SYSTEMS	000070

Assuming for the moment that project IF2000 does not already exist in the dependent table (PROJECT), the following INSERT statement adds a new row with DEPTNO = C01 to PROJECT. This value for DEPTNO must exist in the parent (DEPARTMENT) table.

```
INSERT INTO PROJECT (PROJNO,PROJNAME,DEPTNO,RESPEMP,PRSTAFF)
VALUES ('IF2000','USER EDUCATION','C01','000030',1.00)
```



Applying Update Rules: When a key value is updated, the database manager checks the implicit update rules. A key value may be updated when a parent row (primary key) or a dependent row (foreign key) is updated. If the primary key is updated due to updates made to the parent table, the database manager checks that the updated primary key is unique and is not null. All rows in the dependent table that reference the primary key must first be deleted or updated, or an error will occur. This ensures that the dependent table is not referencing an “old” primary key.

If foreign keys are updated, the database manager checks that each updated foreign key has either a matching primary key in the corresponding parent table, or that the updated foreign key is a null key. A foreign key is null when one or more of its column values are null.

Notes:

1. If a searched update contains a subquery, any table referenced in the subquery must not be a dependent of the table in the UPDATE clause. (See the *DB2 Server for VSE & VM SQL Reference* manual for more information.) In the example below, the NAME table must not be a descendent of the EMPLOYEE table:

```
UPDATE EMPLOYEE
SET SALARY = 65000.00
WHERE LASTNAME = 'SMITH' AND EXISTS
(SELECT * FROM NAME
WHERE LASTNAME = 'SMITH')
```

2. In recoverable storage pools, when a searched update is performed against a column or set of columns, defined in a unique index, primary key, or unique constraint, uniqueness is checked after all rows have been updated. If duplicates exist, then the statement is rolled back.
3. In nonrecoverable storage pools, searched updates are sensitive to the order (ascending or descending) of the data. Since a unique index is automatically created on a primary key column, you cannot use a searched update against a primary key column. This ensures that updates to the primary key are independent of the order of the data.
4. Positioned updates are sensitive to the order (ascending or descending) of the data. Since a unique index is automatically created on a primary key column,

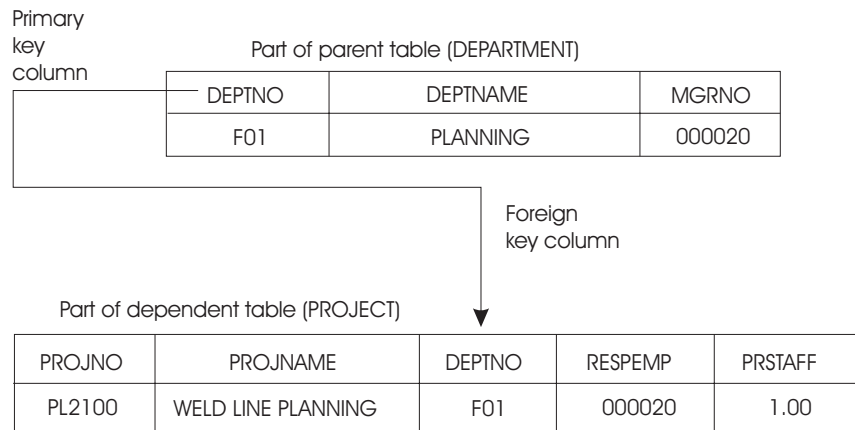
you cannot use a positioned update against a primary key column. This ensures that updates to the primary key are independent of the order of the data.

The following operations change the DEPTNO B01 to F01 in the DEPARTMENT table. Since DEPTNO is a primary key in the parent table, the foreign key with DEPTNO equal to B01 must also be changed in the dependent table (PROJECT).

```
INSERT INTO DEPARTMENT (DEPTNO,DEPTNAME,MGRNO,ADMRDEPT)
VALUES ('F01','PLANNING','000020','A00')
```

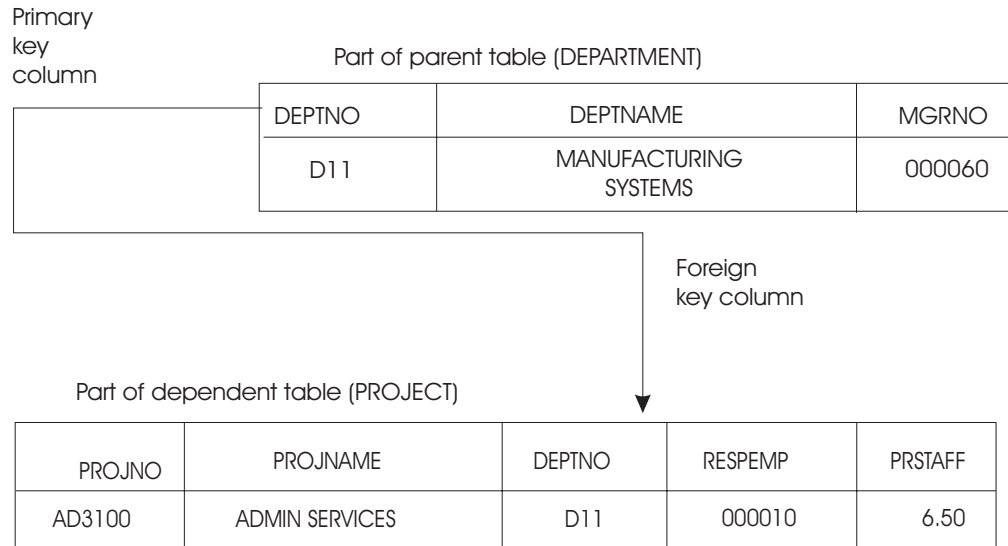
```
UPDATE PROJECT
SET DEPTNO = 'F01'
WHERE DEPTNO = 'B01'
```

```
DELETE FROM DEPARTMENT
WHERE DEPTNO = 'B01'
```



The example below changes the DEPTNO A00 to D11 for the ADMIN SERVICES project in the PROJECT table. Since DEPTNO is a primary key in the parent table, the database manager ensures that DEPTNO D11 in the dependent table (PROJECT) also exists in the parent table (DEPARTMENT).

```
UPDATE PROJECT
SET DEPTNO = 'D11'
WHERE PROJNAME = 'ADMIN SERVICES'
```



Applying Delete Rules: The database manager does not do any checking when data is deleted from dependent tables. The delete rule in a referential constraint clause defines what action should be taken by the database manager when a parent row is deleted. The delete rules are:

- The RESTRICT rule prevents the deletion of a parent row unless all the dependent rows have been deleted first. This is the default rule.
- The SET NULL rule sets all nullable columns of the foreign key to null before deleting the parent row. At least one column of the foreign key must be nullable.
- The CASCADE rule deletes rows at each level containing dependent tables that have the referential constraint CASCADE.

Restrictions on Using Delete Rules

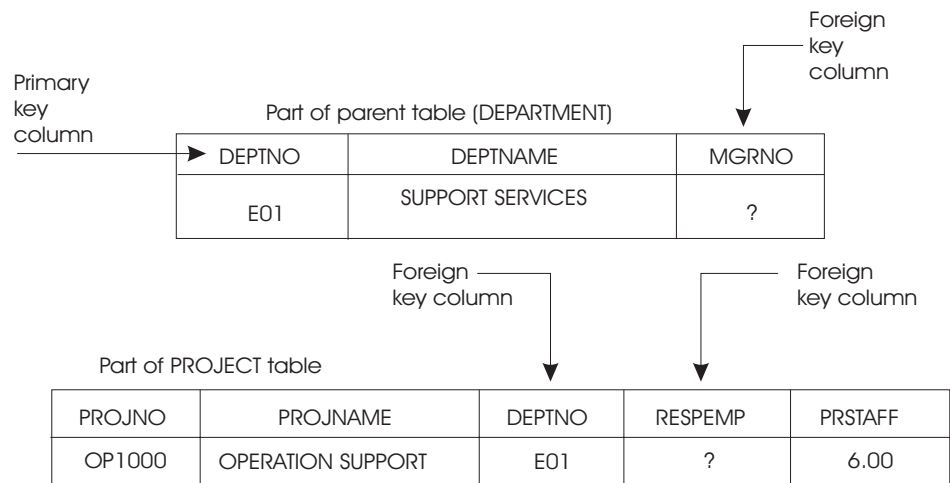
- If a table with a referential constraint of CASCADE has dependent tables that have different delete rules, such as RESTRICT, a delete operation is successful only if the object row is not found in the dependent table. If the object row is found in the dependent table, the CASCADE delete operation is rolled back. That is, the SET NULL and RESTRICT rules maintain their referential integrity between parent and dependent tables.
- A table cannot be delete-connected to itself in a referential cycle involving two or more tables.
- If a dependent table is delete-connected to the parent table through multiple delete paths, each path must have the same delete rule and this rule cannot be SET NULL.
- If a Searched DELETE contains a subquery, any table referenced in the subquery and any table that has a referential constraint of CASCADE or SET NULL with the table referenced in the subquery must not be a dependent of the table in the FROM clause. (See the *DB2 Server for VSE & VM SQL Reference* manual for more information.)

In the following example, the NAME table must not be a descendent of the EMPLOYEE table:

```
DELETE FROM EMPLOYEE
WHERE LASTNAME = 'SMITH' AND EXISTS
(SELECT * FROM NAME
WHERE LASTNAME = 'SMITH')
```

In the example below, the row with EMPNO equal to 000050 is deleted from the EMPLOYEE table:

```
DELETE FROM EMPLOYEE
WHERE LASTNAME = 'GEYER'
```



Because the EMPLOYEE table is a parent table and the delete rule is SET NULL in the relationship that exists between the EMPLOYEE table and the DEPARTMENT table, the database manager sets MGRNO equal 000050 to null in the DEPARTMENT table. Also, because the EMPLOYEE table is a parent table and the delete rule is SET NULL in the relationship that exists between the EMPLOYEE table and the PROJECT table, the database manager sets RESEMP equal 000050 to null in the PROJECT table. (Refer to Figure 84 on page 243 for more information.)

In the example below, the row with DEPTNO equal to D01 is deleted from the DEPARTMENT table:

```
DELETE FROM DEPARTMENT
WHERE DEPTNAME = 'DEVELOPMENT CENTER'
```

Because the DEPARTMENT table is a parent table and the CASCADE rule was set in the relationship that exists between the DEPARTMENT table and the PROJECT table, the row with DEPTNO D01 is also deleted from the PROJECT table.

Generating SQL Statements in Response to Table Modifications

When INSERT, UPDATE, and DELETE statements are issued against tables in a referential structure, the database manager generates internal SQL statements, which it uses to ensure the consistency and integrity of the data in the tables. The number of rows affected, the cost of processing the INSERT, UPDATE, DELETE, and the internally generated statements are returned in the SQLERRD fields in the SQLCA. The SQLERRD(3) gives the number of rows that were processed successfully. Upon successful completion of the DELETE statement, SQLERRD(5) contains the number of dependent rows that were successfully deleted or set to

null. For other data-manipulating language (DML) statements, SQLERRD(5) is set to zero. The relative cost of processing all the statements is given in the SQLERRD(4) field.

Additional information on internally generated statements can be found in tables updated by the EXPLAIN statement. (This statement is discussed in the *DB2 Server for VSE & VM SQL Reference* manual.) To determine this information, enter the EXPLAIN statement for the INSERT, UPDATE, or DELETE statement.

Enforcing Referential Integrity

Referential constraints may be enforced as soon as they are defined, or their enforcement may be deferred. If the constraints are enforced as soon as they are defined, the insert, update, and delete integrity rules are enforced immediately when the INSERT, UPDATE, and DELETE statements are issued.

To defer the enforcement of a constraint is to render the constraint inactive so that it is not immediately enforced when the INSERT, UPDATE, and DELETE statements are issued. This is done by deactivating either the primary key, the dependent foreign key(s), or the foreign key(s). If any of these keys are deactivated, *both* the parent and the dependent tables become inactive and unavailable for data manipulation statements to general users (that is, other than the DBA and the owner of the tables). However, these tables are available for data definition statements.

When a primary key is deactivated, all active dependent foreign keys are implicitly deactivated, and the primary key index is dropped from the parent table. Both parent and dependent tables become inactive. A primary key cannot be implicitly deactivated.

With a table in an inactive state, only the owner of the table or a database administrator (DBA) can enter data manipulating language (DML) statements against it. No one can enter INSERT, UPDATE, and DELETE statements that cause statements to be generated against an inactive table.

When keys (either primary or foreign) are activated, the constraints are automatically verified. If they cannot be verified because of integrity problems, an error message is returned, and the tables remain unavailable for data manipulation statements entered by users other than the DBA or the owner.

When keys (either primary or foreign) are activated or deactivated, packages that depend on the table are invalidated. When the program is run again, it is dynamically reprocessed.

In general, you would defer the enforcement of referential constraints between tables when large amounts of data are to be loaded, or when data is to be loaded in an order that violates the referential constraint at some point during the loading operation. For further information, refer to the *DB2 Server for VSE Database Administration* manual.

The relationships among the EMPLOYEE, DEPARTMENT, and PROJECT tables are shown in Figure 84 on page 243.

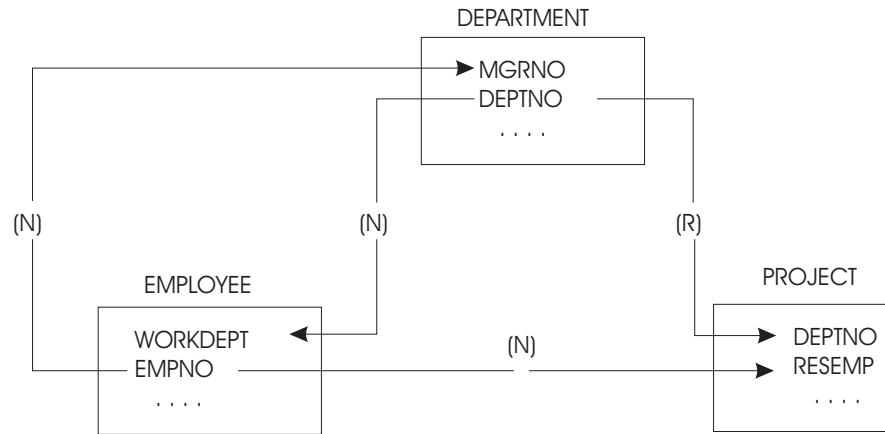


Figure 84. Relationships among the TABLES. Arrows point from primary keys in parent tables to foreign keys in dependent tables. Delete rules are labeled as (C) = CASCADE, (N) = SET NULL, (R) = RESTRICT.

```

CREATE TABLE EMPLOYEE
  (EMPNO      CHAR(6)      NOT NULL, ← primary key
   FIRSTNAME  VARCHAR(12)  NOT NULL,
   MIDINIT    CHAR(1)     NOT NULL,
   LASTNAME   VARCHAR(15)  NOT NULL,
   WORKDEPT   CHAR(3)     ,
   PHONENO    CHAR(4)     ,
   SALARY     DECIMAL(9,2) ,
   PRIMARY KEY (EMPNO)    )

CREATE TABLE DEPARTMENT
  (DEPTNO     CHAR(3)      NOT NULL, ← primary key
   DEPTNAME   VARCHAR(36)  NOT NULL,
   MGRNO      CHAR(6)     , ← foreign key
   PRIMARY KEY (DEPTNO)
   FOREIGN KEY MNUM (MGRNO)
     REFERENCES EMPLOYEE ON DELETE SET NULL)

ALTER TABLE EMPLOYEE ADD FOREIGN KEY WORKNUM (WORKDEPT)
  REFERENCES DEPARTMENT ON DELETE SET NULL

CREATE TABLE PROJECT
  (PROJNO     CHAR(6)      NOT NULL, ← primary key
   PROJNAME   VARCHAR(24)  NOT NULL,
   DEPTNO     CHAR(3)     NOT NULL, ← foreign key
   RESPEMP    CHAR(6)     NOT NULL,
   PRSTAFF    DECIMAL(5,2) ,
   PRIMARY KEY (PROJNO)
   FOREIGN KEY DNUM (DEPTNO)
     REFERENCES DEPARTMENT ON DELETE CASCADE)

```

Then,

```
ALTER TABLE DEPARTMENT DEACTIVATE PRIMARY KEY
```

explicitly deactivates the primary key in DEPARTMENT, and implicitly deactivates the foreign keys DNUM in the PROJECT table and WORKNUM in the EMPLOYEE table. The DEPARTMENT, EMPLOYEE, and PROJECT tables become inactive.

Therefore, only the owner of these tables or the DBA can enter data manipulation statements against the tables.

However,

```
ALTER TABLE DEPARTMENT DEACTIVATE FOREIGN KEY MNUM
```

will not affect the primary key in the EMPLOYEE table. However, both the EMPLOYEE table and the DEPARTMENT table become inactive since the foreign key affects both tables. As mentioned earlier, when tables become inactive, only the owner of the tables or the DBA can enter data manipulation statements against them.

Removing Referential Constraints

To remove a referential constraint, you must drop the foreign key. When a table that contains foreign keys is dropped, the referential constraints associated with that table are removed. You can drop a table explicitly with the DROP TABLE statement, or implicitly with the DROP DBSPACE statement. You can also drop the foreign key with the ALTER TABLE statement, provided that you have the ALTER privilege on the dependent table and the REFERENCES privilege on the parent table. For descriptions of the above three statements, see Chapter 8, “Maintaining Objects Used by a Program” on page 199.

When a table that contains a primary key is dropped, the database manager drops the primary key and any foreign keys that reference the primary key and removes the referential constraints associated with those foreign keys. The ALTER TABLE statement can also be used to drop a primary key directly. To use the ALTER TABLE statement for this purpose, you must have the ALTER and REFERENCES privileges on the parent table as well as the ALTER privilege on all dependent tables.

When keys are dropped, any packages that depend on the table are invalidated. When the program is run again, it is dynamically reprocessed. The new package no longer contains internally generated statements to enforce referential integrity.

Switching Application Servers

You can access multiple application servers from within an application program, but only one application server can be accessed at a time. VSE application servers must reside on the same processor as the user, if DRDA protocol is not being used. If the VSE application server does not reside on the same processor as the CICS/VSE online user, the VSE application server must be accessed during the DRDA protocol. This VSE server must be defined as a remote DRDA server to the DB2 Server for VSE Online Resource Adapter. VM application servers, accessed through VSE guest sharing or using the DRDA protocol, may reside on the same processor as the user, or on another processor (in the TSAF collection or the SNA network). If a program is written to access multiple application servers, its package must exist on all of them.

This section discusses these authorities in more detail, and explains how to switch application servers from your application program. For a detailed discussion on establishing communication links between application requesters and application servers, refer to the *DB2 Server for VSE System Administration* manual.

Identifying Switching Options

Use the CONNECT statement to switch among application servers if you want application programs to connect to different application servers while running. For more information on the CONNECT statement, see the *DB2 Server for VSE & VM SQL Reference* manual.

How to Switch Servers

Figure 85 shows how an application program, indicated by PGM, accesses three application servers by switching application servers in the program. The application servers can reside on the same processor as the program or on a different processor. When you are switching application servers in the program, an application program can switch to a new application server during execution with the CONNECT statement. A package for the program must exist on all application servers it accesses, and each logical unit of work must end before you switch to a different application server. See “Preprocessing the Program on Multiple Application Servers” on page 247 for more details.

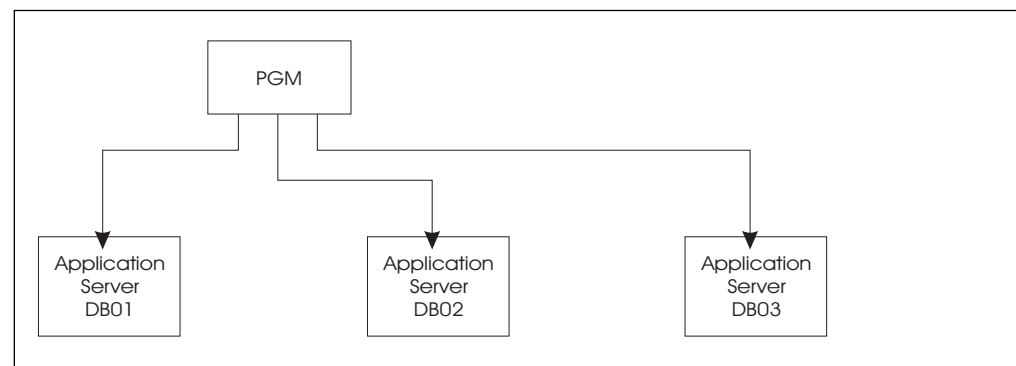


Figure 85. Switching Application Servers Implemented

Accessing a New Application Server

An application accesses the default application server when:

- The first CONNECT statement in an application does not contain the TO clause.
- Either a COMMIT RELEASE or ROLLBACK RELEASE statement is executed by a batch application and the subsequent CONNECT statement does not contain the TO clause.
- No CONNECT statement is executed by a CICS/VSE application. That is, an implicit connect is performed.

For more information about the defaults that determine the application server that is accessed, refer to the *DB2 Server for VSE System Administration* manual.

You must enter a CONNECT statement from a batch application after a COMMIT RELEASE statement or ROLLBACK RELEASE statement to reestablish the user ID and target application server. Otherwise subsequent SQL statements are not successful (SQLCODE -563). A null CONNECT statement is not sufficient.

To query the user ID and the identity of the application server to which you are currently connected, enter one of the following from within an application program:

- A null CONNECT statement, which returns the user ID and the identification of the application server in the SQLCA. Refer to the discussion of the CONNECT statement in the *DB2 Server for VSE & VM SQL Reference* manual for a description of the format and location of the information that is returned.

If a null CONNECT is issued as the first SQL statement in a batch application, blanks are returned in the SQLCA for the user ID and application server name and the execution of subsequent SQL statements are not successful (SQLCODE -563).
- A SELECT statement requesting the USER and CURRENT SERVER, which returns the user ID and the identification of the application server in the host variables associated with the USER and CURRENT SERVER special registers.

Illustrating Sample Code

Figure 86 shows how an application can take advantage of switching application servers.

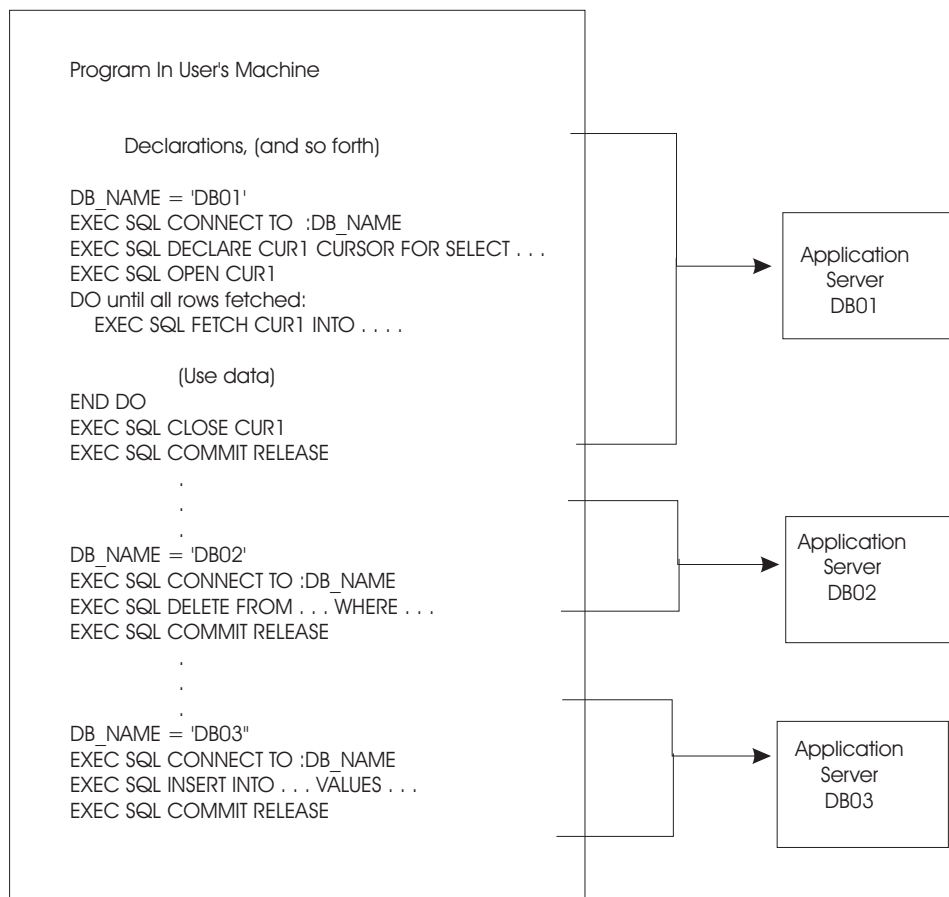


Figure 86. Pseudocode Illustrating How to Switch Application Servers

In the above example, the application connects to three application servers (DB01, DB02, and DB03), and performs a series of operations when accessing each one. When accessing DB01, the program retrieves information from the application server (with the FETCH statement) and processes the information.

Next, it accesses DB02, and some rows are deleted from a table; then accesses DB03, and rows are inserted into a table.

Preprocessing the Program on Multiple Application Servers

An application program that allows access to multiple application servers with the CONNECT statement must exist on every application server that the program is to access. The preprocessors provide the DBNAME parameter to preprocess a program on different application servers. In addition, the CBND transaction provides the DBLIST parameter to create a package on different application servers.

When an application that accesses different application servers is being preprocessed, certain warnings may be issued by the preprocessor. For example, if TABLE1 exists in DB01, but your application program is preprocessed against DB02 or DB03, you will receive warning messages that the table does not exist in those application servers. If your program does not access TABLE1 in DB02 or DB03, these messages can be ignored; however, if TABLE1 will be accessed in either DB02 or DB03, you must create TABLE1 in the accessed application server.

You should reprocess the program on the application servers that you updated before executing the program. If you are using the preprocessing option CTOKEN=NO, you only need to preprocess the application program on one application server. If you specify CTOKEN=YES, you must reprocess on all application servers that the program accesses to get the same timestamp.

During execution, the table being referenced in an SQL statement may reside in the currently accessed application server or in another application server. In fact, a table of the same name, but with different attributes, may be in the application server. The database manager issues a warning message that there are inconsistencies, but preprocessing will continue. The statement causing the warning remains in the package, and will only cause an application failure if it is referenced at run time. Conditions that will generate a warning and the corresponding SQLCODE include:

- Column *column* was not found in table *owner.table*. (SQLCODE = +205 and SQLSTATE='01533')
- Incompatible data types were found in an expression or compare operation. (SQLCODE = +401 and SQLSTATE='01578')
- The string representation of a date/time value has invalid syntax. (SQLCODE = +180 and SQLSTATE='01572')

Condition Handling with LE/VSE

The DB2 Server for VSE environment is sensitive to errors or conditions. A failing SQL transaction or application can potentially leave a DB2 Server for VSE database in an inconsistent state. For this reason, it is essential that DB2 Server for VSE knows about the failure of a transaction or application that has been updating a database so that it can perform database rollback.

When a user runs an application with the TRAP(ON) run-time option of LE/VSE and the DB2 Server for VSE application is running in Single User Mode, LE/VSE and DB2 Server for VSE keep track of calls to and returns from the database. If a program interrupt or abend occurs when the application is running, the LE/VSE condition manager is informed whether the problem occurred in the application or in the database manager. If the program interrupt or abend occurs in the database manager, the LE/VSE condition handler passes the condition back to DB2 Server for VSE.

If a program interrupt or abend occurs in the application outside the database manager, the LE/VSE condition manager will perform its own condition handling actions. If the condition manager gets control then the user **must** do one of the following:

- Resolve the error completely so that the application can continue.
- Make sure that the application terminates abnormally by using the ABTERMENC(ABEND) run-time option of LE/VSE to transform all abnormal terminations into operating system abends in order to cause DB2 Server for VSE to do the necessary recovery processing when the DB2 Server for VSE server is warm started.

Note: The following methods are available for specifying any LE/VSE run-time options, including ABTERMENC(ABEND):

1. As an installation wide default through the CEEDOPT assembler language source file.
2. In the assembler user exit routine CEEBXITA.
3. As an application default through the CEEUOPT assembler language source file. CEEUOPT is assembled into an object module which is linked with the application program.
4. In JCL through the PARM parameter of the JCL EXEC statement.
5. In PL/I source code through the PLIXOPT string.

See the *IBM Language Environment for VSE/ESA Programming Guide* for more details.

- Provide a modified run-time assembler user exit (CEEBXITA) that transforms all abnormal terminations into operating system abends. The assembler user exit should check the return code and reason code or the CEEAUE_ABTERM bit, and request an abend by setting the CEEAUE_ABND flag to ON, if appropriate.

Note: CEEBXITA assembler user exit is intended for use by the application programmer. It is not intended for DB2 Server for VSE use. See the *IBM Language Environment for VSE/ESA Programming Guide* for more details.

Appendix A. Using SQL in Assembler Language

Contents

Using ARIS6ASD, an Assembler Language Sample Program	250
Acquiring the SQLDSECT Area	250
Imposing Usage Restrictions on the SQLDSECT Area	251
Rules for Using SQL Statements in Assembler Language	253
Identifying Rules for Case	253
Declaring Host Variables	253
Embedding SQL Statements	255
Using the INCLUDE Statement	256
Using Host Variables in SQL Statements	256
Using DBCS Characters in Assembler Language	256
Handling SQL Errors	256
Using Dynamic SQL Statements in Assembler Language	257
Defining DB2 Server for VSE Data Types for Assembler Language	258
Using Reentrant Assembler Language Programs	260
Using Stored Procedures	265

Using ARIS6ASD, an Assembler Language Sample Program

ARIS6ASD is an assembler language sample program for VSE systems that is shipped with the DB2 Server for VSE product. It resides on the production disk for the base product. You may find it useful to print this sample program before going through this appendix as the hard copy will provide an illustration for many of the topics discussed here.

Note, for example, how the program satisfies the requirements of the application prolog and epilog. Near the beginning of the program, all the host variables are declared, the SQLDSECT area is acquired (and set to zero), and error handling is defined. Near the logical end of the program, the database changes are rolled back, to assure that the database remains consistent for each use of the sample program. (For your own applications, of course, you will enter a COMMIT statement.)

The DS and DC statements for the host variables were determined by referring to Figure 91 on page 258, which shows the assembler representation for each of the DB2 Server for VSE data types supported by assembler programs. When you are coding your own applications, you must obtain the data types of the columns that your host variables interact with. This can be done by querying the catalog tables. These tables are described in the *DB2 Server for VSE & VM SQL Reference* manual.

Acquiring the SQLDSECT Area

The assembler preprocessor puts all the variables and structures it generates within a DSECT named SQLDSECT. The preprocessor also generates a fullword variable called SQLDSIZ, which contains the length of the SQLDSECT DSECT in bytes. Thus, for all assembler programs, you must provide an area of size SQLDSIZ, set the area to zero, and provide addressability to the SQLDSECT DSECT.

Figure 87 on page 251 shows sample code that does just that for VSE batch and ICCF applications:

```

TESTNAME CSECT
        STM 14,12,12(13)
        BALR regx,0
        USING *,regx
        L 0,SQLDSIZ
        GETVIS ADDRESS=(1),LENGTH=(0)
        LR regy,1
        USING SQLDSECT,regy
        (add code to zero the area)
        .
        .
        .
        END

```

This area is needed only until the program is finished executing all SQL statements, at which time the area should be freed (FREEVIS).

Figure 87. Acquiring the SQLDSECT Area for VSE Batch and ICCF Applications

If you know the approximate size of the SQLDSECT that will be generated in your program, you can define an area (AREA DS CLxxx) within your program and use this as your SQLDSECT area. Your program will not be re-entrant if you use this method.

The preprocessor generates the code to calculate SQLDSIZ directly in front of the last statement in the source program. Make the last statement an END statement.

If the assembler preprocessor is run with the CHECK option, SQLDSECT and SQLDSIZ are not generated. Errors occur if you attempt to assemble the output generated by the preprocessor when the CHECK option is specified. See Chapter 4, "Preprocessing and Running a Program" on page 105 for more information about for more information about preprocessor parameters.

For CICS/VSE transactions, Figure 87 does not apply. Figure 88 is a CICS/VSE example.

```

label1 EQU regx
EXEC CICS GETMAIN SET(label1) LENGTH(SQLDSIZ+2) INITIMG(00)
USING SQLDSECT,regx

```

Figure 88. Acquiring the SQLDSECT Area for CICS/VSE Applications

Note: You must provide a save area for all assembler programs.

Imposing Usage Restrictions on the SQLDSECT Area

There are two performance considerations about the SQLDSECT area that you should be aware of:

- Acquire and clear the SQLDSECT area only once.

The examples shown in Figure 87 and Figure 88 assume that the TESTNAME is entered once. If TESTNAME is a subroutine of a mainline module, and if TESTNAME is invoked many times, you should acquire the SQLDSECT in the mainline module. The following is an example of how this may be done:

1. In TESTNAME add an entry card as follows:

```
ENTRY SQLDSIZ
```

This allows the field containing the size information for the SQLDSECT area to be accessed externally.

2. The mainline module can now access the size information using the following sequence:

```
L regy,=V(SQLDSIZ) GET POINTER TO FIELD CONTAINING SIZE
L 0,0(,regy) SET LENGTH
GETVIS ADDRESS=(1),LENGTH=(0)
LR regy,1 SAVE POINTER TO SQLDSECT
(Zero the SQLDSECT area.)
```

3. When the mainline module calls TESTNAME, it should pass the pointer to the SQLDSECT. Assuming that regy still contains the pointer, TESTNAME simply issues the appropriate USING statement as follows:

```
TESTNAME CSECT
STM 14,12,12(13)
BALR regx,0
USING SQLDSECT,regy
.
.
```

Depending on how many times TESTNAME is invoked, the above could be an important performance consideration. Using the technique reduces the path length because you only need to get, clear, and free storage once. Further, the cleared SQLDSECT area serves as a “first pass” flag for the batch/ICCF and CMS resource adapters. Thus, by letting the mainline module initialize the SQLDSECT area only once, you further avoid significant resource adapter “first pass” processing.

- Provide only one SQLDSECT area.

If you structure an application so that the mainline module invokes several modules that each contain SQL commands, you need to provide only one SQLDSECT area. The area that you provide must be the largest SQLDSECT area. For example, suppose the mainline module invokes MODA and MODB, each of which contains SQL commands, but which have different SQLDSECT area requirements. The mainline module must satisfy the larger of the two requirements.

By inserting the following into MODA and MODB, you could allow the mainline module to calculate the SQLDSECT area requirement:

```
INTO MODA:                               INTO MODB:

MODADSIZ DC A(SQLDSIZ)                   MODBDSIZ DC A(SQLDSIZ)
        ENTRY MODADSIZ                   ENTRY MODBDSIZ
        .                                 .
        .                                 .
```

The mainline module could reference the above entries and provide for the maximum SQLDSECT area. The following example shows how the mainline module could determine the requirement of MODA:

```

L regy,=V(MODADSIZ)  GET POINTER TO POINTER FIELD
L regy,0(,regy)      GET POINTER TO FIELD CONTAINING SIZE
L 0,0(,regy)         SET LENGTH.

```

The same technique could be used to access the SQLDSIZ of MODB. Given the two SQLDSIZ values, the mainline module should provide for a SQLDSECT area equal in size to the greater SQLDSIZ value.

By using only one SQLDSECT area for your application, you reduce the storage requirement and minimize the first pass processing.

Rules for Using SQL Statements in Assembler Language

This section lists the rules for embedding SQL statements within an assembler program.

Note: OPSYN and ICTL assembler statements may not be used.

Identifying Rules for Case

Uppercase must be used for all SQL statements, except for text within quotation marks, which will be left in the original case.

Declaring Host Variables

The following example shows an SQL declare section for an assembler program:

Col. 1	Col.16	Col. 72
LABEL EXEC SQL	BEGIN DECLARE SECTION	
AA DS F		
BB DC H'3'	comment	
* comment card or		
* comment section		
CC DC	CL80'xxxx.....xxxx*	
	xxxx.....xxxxx'	
XYZ DSECT		
DD DS D		
EE DS CL5		
FF DS H,CL40		
	ORG FF	
GG DS H		
HH DS CL40	comment	
*	continued comment	
II DS PL5		
JJ DC PL5'123.45'		
KK DS 0H		
LL DS CL12		
XX DS CL10		*
	continuation of comment	
LABEL2 EXEC SQL	END DECLARE SECTION	comment

The preceding example illustrates the following rules:

1. All assembler variables that are to be used in SQL statements must be declared, and their declarations must appear within one or more sections that begin with:

```
EXEC SQL BEGIN DECLARE SECTION
```

and end with:

```
EXEC SQL END DECLARE SECTION
```

Each of these two statements must be totally contained on one line.

Note: There is no semicolon delimiter at the end of the SQL statements. There may be a label on either of the statements, and host language comments are allowed after the statements.

2. Host language comments are allowed on any statement within the SQL declare section, as are host language comment line images (* in column 1).
3. The assembler preprocessor processes the statements in the declare section as follows:
 - a. If there is no label, the preprocessor ignores the statement and goes on to the next.
 - b. If there is a label, but the opcode is not DS or DC, the preprocessor ignores the statement and goes on to the next.
 - c. If there is a label and a DS or DC opcode, the operand is checked. The operand must be an acceptable data type, as shown in Figure 91 on page 258. Here are some examples:

```
F
F'5'
H
H'100'
CL255
CL5'ABCDE'
H,CL5
H'5',CL5'ABCDE'
D
D'2.5E10'
PL2
PL5'123.45'
P'123'
P'123.45'
P'1234'
P'123.456'
H,CL32767
```

The first character of the operand may also be zero and used as follows:

```
0H
0F
0D
0C
```

In this case, the line is ignored and the next line is processed.

If there are no errors at this stage, the variable is validly defined as a host variable. If there are errors, the line is flagged as an error, and the next line is processed.

4. The database manager allows host variable names, statement labels, and SQL descriptor area names of up to 256 characters in length, subject to any assembler language restrictions mentioned in this appendix.
5. The opcode for a declare statement must be coded on the first line of the statement. Because the line length is 71, this limits the length of host variable names to 68 characters.
6. Continuations are allowed by coding a non-blank character in column 72 of the line to be continued, and coding the continuation anywhere from columns 16 to 71 inclusive on the next line, leaving 1-15 blank.
7. Continuation of *tokens* (the basic syntactical units of a language) is allowed from one line to the next, by coding the first part of the token up to column 71 of the line to be continued, and coding the second part of the token from column 16 on the continuation line. If either column 71 of the continued line or column 16 of the continuation line is blank, the token will not be continued. See the *DB2 Server for VSE & VM SQL Reference* manual for a discussion on tokens.
8. The declare section can be anywhere that a normal DS or DC can be used. Because the assembler preprocessor is a two-pass operation, the declare section can come after the SQL statements that use the host variables.
9. There can be more than one SQL declare section in a program.
10. Host variable names cannot contain variable symbols (for example, &ABCDEFG, &SYSNDX, &SYSPARM). These names must be resolved at preprocessing time. Variable symbols will be resolved at assembly time.

Embedding SQL Statements

The following are the rules for embedding SQL statements within assembler programs:

1. Each SQL statement must be preceded by EXEC SQL, which must be on the same line. Only blanks can appear between the EXEC and SQL. There must not be a semicolon (;) delimiter on the SQL statement.
2. The first line of an SQL statement can have a label beginning in column 1. If there is no label, the statement must begin in column 2 or greater.
3. Rules for continuation of statements and tokens are the same as those described for host variables.
4. No host language comments are allowed within an SQL statement. Any such comments are considered part of the SQL statement.
5. If an entire statement must be contained on one line, there cannot be SQL comments embedded in the statement. There are three such statements:
 - BEGIN DECLARE SECTION
 - END DECLARE SECTION
 - INCLUDE.
6. Avoid using labels or variable names that begin with SQL, ARI, or RDI. Also avoid names beginning with PID, PBC, PA, PB, PC, PD, PE, PL, or PN where these letters are followed by numbers. These names may conflict with names generated by the assembler preprocessor.
7. All SQL statements must be in one CSECT.

- The EXEC SQL must be coded on the first line of the statement. Because the line length is 71, this limits the length of a LABEL to 62 characters.

Using the INCLUDE Statement

To include external secondary input, specify the following at the point in the source code where the secondary input is to be included:

```
EXEC SQL INCLUDE text_name
```

Text_name is the A-Type source member of a VSE library, is the file name of a CMS file with an "ASMCOPY"

The INCLUDE statement must be completely contained on one line. There may be a label on the command, and host language comments are allowed after the command.

Using Host Variables in SQL Statements

When you place host variables within an SQL statement, you must put a colon (:) in front of every host variable, to distinguish them from the SQL identifiers (such as a column name). When the same variable is used outside of an SQL statement, do not use a colon.

A host variable can represent a data value, but not an SQL identifier. For example, you cannot assign a character constant, such as "MUSICIANS", to a host variable, and then use that host variable in a CREATE TABLE statement to represent the table name. The following pseudocode sequence is invalid:

```
IT = 'MUSICIANS'  
CREATE TABLE :TT (NAME ...
```

Incorrect

Using DBCS Characters in Assembler Language

The rules for the format and use of DBCS characters in SQL statements are the same for assembler language as for other host languages supported by the database manager. For a discussion of these rules, see "Using a Double-Byte Character Set (DBCS)" on page 50.

Assembler language does not provide a way to define graphic host variables. If you want to add graphic data to or retrieve it from DB2 Server for VSE tables, you must execute the affected statements dynamically. By doing so, the data areas that are referenced by each statement can be described in an SQLDA. In the SQLDA, you must set the data type of the areas containing graphic data to one of the graphic data types. For a discussion of the SQLDA, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

Handling SQL Errors

There are two ways to declare the SQL communication area (SQLCA):

- You can code the following statement in your source program:

```
EXEC SQL INCLUDE SQLCA
```

The preprocessor replaces this with a declaration of the SQLCA structure.

- You may declare the SQLCA directly, as shown in Figure 89 on page 257.

SQLCA	DS	0F
SQLCAID	DS	CL8
SQLCABC	DS	F
SQLCODE	DS	F
SQLERRM	DS	H,CL70
SQLERRP	DS	CL8
SQLERRD	DS	6F
SQLWARN	DS	0C
SQLWARN0	DS	CL1
SQLWARN1	DS	CL1
SQLWARN2	DS	CL1
SQLWARN3	DS	CL1
SQLWARN4	DS	CL1
SQLWARN5	DS	CL1
SQLWARN6	DS	CL1
SQLWARN7	DS	CL1
SQLWARN8	DS	CL1
SQLWARN9	DS	CL1
SQLWARNA	DS	CL1
SQLSTATE	DS	CL5

Figure 89. SQLCA Structure (in Assembler)

You must not declare the SQLCA within the SQL declare section. The meaning of the fields is explained in *DB2 Server for VSE & VM SQL Reference* manual.

You may find that the only variable in the SQLCA you really need is SQLCODE. If this is the case, declare just the SQLCODE variable, and invoke NOSQLCA support at preprocessor time.

Using Dynamic SQL Statements in Assembler Language

An SQLDA structure may be required for dynamically executed SQL statements. There are two ways to declare the SQLDA structure:

- You can code the following statement in your source program:

```
EXEC SQL INCLUDE SQLDA
```

The preprocessor replaces this with a declaration of the SQLDA structure.

- You can declare the SQLDA directly, as shown in Figure 90 on page 258.

```

SQLDA      DSECT
SQLDAID   DS      CL8
SQLDABC   DS      F
SQLN      DS      H
SQLD      DS      H
SQLVAR    DS      0F
SQLVARN   DSECT
SQLTYPE   DS      H
SQLLEN    DS      0H
SQLPRSCN  DS      CL1
SQLSCALE  DS      CL1
SQLDATA   DS      A
SQLIND    DS      A
SQLNAME   DS      H,CL30
&SYSECT   CSECT

```

Figure 90. SQLDA Structure (in Assembler)

The SQLDA structure must not be declared within an SQL declare section. When you specify INCLUDE SQLDA, the assembler preprocessor generates a CSECT statement at the end of the SQLDA. This CSECT is generated with the name of the CSECT currently active in your program.

You must not specify a constant string on a PREPARE or EXECUTE IMMEDIATE statement. You can only specify a host variable defined as a variable-length character string:

```

EXEC SQL PREPARE S1 FROM :STRING1
EXEC SQL EXECUTE IMMEDIATE :STRING1
      .
      .
EXEC SQL BEGIN DECLARE SECTION
STRING1 DS H,CLxxxxx (xxxxx <= 8192)
EXEC SQL END DECLARE SECTION

```

The halfword of STRING1 must contain the length of the string, and the character portion must contain the string itself when the PREPARE or EXECUTE IMMEDIATE statement is executed.

See Appendix B of the *DB2 Server for VSE & VM SQL Reference* manual for more information on the individual fields within SQLDA.

Defining DB2 Server for VSE Data Types for Assembler Language

Figure 91 (Page 1 of 3). DB2 Server for VSE Data Types for Assembler

Description	DB2 Server for VSE Keyword	Equivalent Assembler Declaration
A binary integer of 31 bits, plus sign.	INTEGER or INT	F
A binary integer of 15 bits, plus sign.	SMALLINT	H

Figure 91 (Page 2 of 3). DB2 Server for VSE Data Types for Assembler

Description	DB2 Server for VSE Keyword	Equivalent Assembler Declaration
<p>A packed decimal number, precision p, scale s ($1 \leq p \leq 31$ and $0 \leq s \leq p$). In storage the number occupies a maximum of 16 bytes. Precision is the total number of digits. Scale is the number of digits to the right of the decimal point.</p>	<p>DECIMAL[(p,s)] or DEC[(p,s)](1) ¹</p>	<p>PLn['decimal constant'] or P'decimal constant'</p> <p>For declarations using PLn, the precision is $2n-1$ (n is the number of bytes). For the declarations using P, the length of the decimal constant, excluding the decimal point and sign, is the precision. For the declarations using P or PL, the scale is that of the decimal constant. For the declarations using P, the decimal constant must be specified. For the declarations using PLn, the decimal constant is optional. If it is not specified, the scale is 0.</p>
<p>A single precision (4-byte) floating-point number in short System/390 floating-point format.</p>	<p>REAL or FLOAT(p), $1 \leq p \leq 21$</p>	<p>E</p>
<p>A double precision (8-byte) floating-point number in long System/390 floating-point format.</p>	<p>FLOAT or FLOAT(p), $22 \leq p \leq 53$ or DOUBLE PRECISION</p>	<p>D</p>
<p>A fixed-length character string of length n where $0 < n \leq 254$.</p>	<p>CHARACTER[(n)] or CHAR[(n)]</p>	<p>CLn</p>
<p>A varying-length character string of maximum length n. If $n > 254$ or $\leq 32,767$; this data type is considered a long field. (See "Using Long Strings" on page 45.) (Only the actual length is stored in the database.)</p>	<p>VARCHAR(n)</p>	<p>H,CLn</p>
<p>A varying-length character string of maximum length 32 767 bytes.</p>	<p>LONG VARCHAR</p>	<p>H,CLn</p>
<p>A fixed-length string of n DBCS characters, where $0 < n \leq 127$.</p>	<p>GRAPHIC[(n)]</p>	<p>Not supported.</p>
<p>A varying-length string of n DBCS characters. If $n > 127$ or ≤ 16383, this data type is considered a long field. (See "Using Long Strings" on page 45.)</p>	<p>VARGRAPHIC(n)</p>	<p>Not supported.</p>
<p>A varying-length string of DBCS characters of maximum length 16 383.</p>	<p>LONG VARGRAPHIC</p>	<p>Not supported.</p>
<p>A fixed or varying-length character string representing a date. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.</p>	<p>DATE</p>	<p>CLn or H,CLn</p>
<p>A fixed or varying-length character string representing a time. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.</p>	<p>TIME</p>	<p>CLn or H,CLn</p>

Figure 91 (Page 3 of 3). DB2 Server for VSE Data Types for Assembler

Description	DB2 Server for VSE Keyword	Equivalent Assembler Declaration
A fixed or varying-length character string representing a timestamp. The lengths can vary on input and output. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIMESTAMP	CLn or H,CLn

Notes:

1. NUMERIC is a synonym for DECIMAL, and may be used when creating or altering tables. In such cases, however, the CREATE or ALTER function will establish the column (or columns) as DECIMAL.

Using Reentrant Assembler Language Programs

A reentrant program has the characteristic of dynamic allocation of space for data and save areas. This reentrant characteristic can be used in assembler programs. In this case, the data and save areas are allocated in a calling (driver) program and passed to a called (reentrant) program as parameters. Storage for these areas need not be allocated in the called program.

A convenient use for reentrancy is the use of an SQLDA structure declared as a DSECT in the calling program. This, in combination with an INCLUDE SQLDA statement in the called program, permits the passing back of values, extracted by a SELECT/FETCH in the called program, in a clean and simple manner. A DESCRIBE statement can be used by the called program to fill the SQLDA structure, or it can be hand-filled in the driver program. Other SQL statements (for example, INSERT, DELETE, UPDATE) utilize a single data location to communicate just an SQLCODE.

If statement results other than the SQLCODE are desired, an SQLCA structure can be allocated in the driver program. However, unlike the SQLDA structure allocation by a DSECT, the fields of the SQLCA structure must be hard-coded into the driver, because the driver will not be preprocessed. An INCLUDE SQLCA statement, within a DSECT, is then required in the called program. SQLCA communication between the two programs can be achieved by passing the address of the first field of the SQLCA structure to the reentrant program.

The "Locda DSECT" structure is hard-coded in the Driver Program, instead of being defined by an "EXEC SQL INCLUDE SQLDA", so that there is no need to preprocess the Driver Program. This example assumes there is only a single host variable returned by the FETCH. For production application programming, it is recommended that macros be created for defining the SQLCA and SQLDA structures (with optional DSECT statement) when used in programs that will *not* be preprocessed.

The following are skeleton programs illustrating the use of the SQLDA structure, and a single data location for communicating SQLCODEs. The reentrant example illustrates only a FETCH statement. If more than one "action" statement (INSERT, DELETE, and so on) is used, then various flags are needed to direct access to the

individual operations. The required modifications to include an SQLCA structure follow these skeletons.

```

Driver  CSECT ,           Driver Program
* Standard Linkage Conventions ...
      STM  R14,R12,12(R13)   Save callers registers
      :
Qstring DC  H'57',CL57'SELECT DESCRIPTION FROM INVENTORY WHERE QONHA $
          ND < 100'         SQL Statement to be executed
      :
      LA  R13,Save1         Subroutine Register Savearea Address
* Forward and backward chain saveareas together
      :
      LA  R4,1              '1' indicates 1st call to subroutine
      ST  R4,Loccode        SQLCODE returned from subroutine
*                               (Also used as 1st call switch)
      :
* Create SQLDA structure to pass to subroutine:
* (OR Subroutine could fill in by using DESCRIBE)
      LA  R4,LSQLDA         Point R4 at SQLDA area
      USING Locda,R4        reference SQLDA fields
      :
      LA  R7,Outarea+1     Address where DESCRIPTION stored
      ST  R7,Locdata
      LA  R7,Indaddr       Address where Indicator Value stored
      ST  R7,Locind
* NOTE: Setting of other SQLDA fields is not shown, but may be required
      :
*
* Loop to call reentrant subroutine (Loop needed for Cursor operation)
LOOP    EQU  *
*
* Blank Output area for next FETCH result:
      :
      LA  R1,Parmlist      Parms passed to subroutine through R1
      L   R15,=V(Reentran) Load Subroutine Entry Point address
      BALR R14,R15         Call Reentrant Subroutine
      CLC  Loccode,F0      Any error from subroutine ?
      BE  FetchOK         No, continue as normal
      CLC  Loccode,F100    Cursor EOF occurred ???
      BE  Final           Yes, all done.
      B   Errchk          No, some kind of error, go handle.
*
FetchOK EQU  *
* Test indicator values for NULL, etc, and handle as appropriate:
      :
*
* Output result from a Fetch: (Data conversion may be necessary)
      :
*
* Branch back to Loop for another Fetch
      B   LOOP
      :
Errchk  EQU  *
* Handle errors returned by subroutine.
      :
Final   EQU  *
* Program complete, restore registers and return to caller
      :
      BR  R14              Return to caller
      :

```

Figure 92 (Part 1 of 2). Driver Program

```

* Declare Section
:
:
F0      DC    F'0'          'NO ERRORS' retcode from subroutine
F100    DC    F'100'       'CURSOR EOF' retcode from subroutine
:
SaveRA  DS    18F          register savearea for use by Resource
*      ... Adapter when called by subroutine
Save1   DS    18F          subroutine register savearea
Loccode DS    F            SQLCODE variable passed to subroutine
*      (return code from subroutine)
:
Parmlist DS    0D          Subroutine Parameter List:
      DC    A(Qstring)     SQL Statement to execute
      DC    A(LSQLDA)      Local SQLDA area
      DC    A(Loccode)     Return Code from subroutine
      DC    A(Hostvar)     Host Variable Workarea
      DC    A(SaveRA)      Resource Adapter register savearea
:
Indaddr DS    F            Indicator area
Outarea DS    CL80         Fetch value return area
:
LSQLDA  DS    CL500        Local SQLDA area
Hostvar  DS    CL500        Subroutine Host Variable workarea
:
Locda    DSECT ,           Describes SQLDA fields
Locdaid  DS    CL8
Locdabc  DS    F
Locn     DS    H
Locd     DS    H
Locvar   DS    0F          assumes one one Host Variable used
Loctype  DS    H
Loclen   DS    0H
Locprcsn DS    X
Locscale DS    X
Locdata  DS    A
Locind   DS    A
Locname  DS    H,CL30      ...end of Local SQLDA area
:
:
      END Driver          ...end of Driver Program

```

Figure 92 (Part 2 of 2). Driver Program


```

Reentran CSECT ,           Reentrant Subroutine
* Standard Linkage Conventions. Register Savearea address in R13.
  STM  R14,R12,12(R13)    Save callers registers
  :
* Get Parameter addresses
  L    R3,0(0,R1)         Point to Qstring
  L    R4,4(0,R1)         Point to SQLDA area
  USING SQLDA,R4          Reference SQLDA fields
  L    R5,8(0,R1)         Point to Loccode (SQLCODE) return code
  USING LSQCODE,R5        Reference Passed SQLCODE variable
  L    R6,12(0,R1)        Point to Hostvar workarea
  USING Hostvar,R6        Reference Hostvar workarea
  LR   R7,R13             R7 points to callers savearea
  L    R13,16(0,R1)       Point R13 at "our" passed savearea ...
*                               ... for use by Resource Adapter calls
* Forward and backward chain saveareas together
  ST   R13,8(0,R7)        Caller savearea points to "our" savearea
  ST   R7,4(0,R13)        "our" savearea points to caller savearea
  :
* Check if this is first call to subroutine:
  CLC  F0,0(R5)           If NOT zero, it is first call
  BE   Next               Is zero - NOT first call
  EXEC SQL CONNECT ...
  :
  LH   R1,0(R3)           Get length of Qstring
  LA   R1,1(R1,0)         Length minus 1 for EXecute ...
*                               ... plus 2 for length Halfword ...
*                               ... equals length + 1.
  EX   R1,MOVQSTR          move length & Qstring to Hostvar area
  EXEC SQL PREPARE S1 FROM :QSTRING
  CLC  SQLCODE,F0         Any errors ?
  BNE  Exit               Yes, return it to caller
  :
* Fill in passed SQLDA structure (possibly with DESCRIBE),
* if not done in Driver program.
  :
  EXEC SQL DECLARE C1 CURSOR FOR S1
  :
  EXEC SQL OPEN C1
  :
Next  EQU  *
  EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA
  CLC  SQLCODE,F100       Cursor EOF reached ??
  BNE  Exit               No, return to caller (even if error)
  :
*
* All Fetched, Close Cursor before returning
Done  EQU  *
  EXEC SQL CLOSE C1
  CLC  SQLCODE,F0         Any error ?
  BNE  Exit               Yes, return error to caller
*
* Return 'CURSOR EOF' return code to caller
  MVC  0(4,R5),F100      R5 points to Loccode
  :
Exit  EQU  *              Return to caller
  :

```

Figure 93 (Part 1 of 2). Reentrant Program

```

* Restore registers and return to caller
* Our return code is in Loccode
      L   R13,4(0,R13)      Load callers savearea address
      LM  R14,R12,12(R13)   Restore callers registers
      BR  R14               Return to caller
      :
* Declare section
MOVQSTR MVC  QSTRING(1),0(R3) Executed during 1st call
F0      DC  F'0'           'NO ERRORS' retcode from subroutine
F100    DC  F'100'        'CURSOR EOF' retcode from subroutine
      :
* Include the SQLDA DSECT
      EXEC SQL INCLUDE SQLDA
      :
Hostvar DSECT ,           Passed Host Variable Workarea
QSTRING DS  CL500
      :
LSQLCODE DSECT ,         Passed SQLCODE variable
SQLCODE DS  F
      :
      END Reentran      ...end of Reentrant Subroutine

```

Figure 93 (Part 2 of 2). Reentrant Program

To include full SQLCA communications between the Driver Program and the Reentrant program, you must modify both programs.

In the Driver program, replace the "Loccode" variable definition with an "SQLCA" structure definition and update the 3rd address constant in the "Parmlist", as follows:

```

:
Save1   DS   18F           subroutine register savearea
Locca   DS   0D           SQLCA structure passed to subroutine
Loccaid DS   CL8
Loccabc DS   F
Loccode DS   F           SQLCODE
Locerrm DS   H,CL70
Locerrp DS   CL8
Locerrd DS   6F
Locwarn DS   0C
Locwarn0 DS  CL1
Locwarn1 DS  CL1
Locwarn2 DS  CL1
Locwarn3 DS  CL1
Locwarn4 DS  CL1
Locwarn5 DS  CL1
Locwarn6 DS  CL1
Locwarn7 DS  CL1
Locwarn8 DS  CL1
Locwarn9 DS  CL1
LocwarnA DS  CL1
Locstate DS  CL5         ... end of local SQLCA structure
:
Parmlist DS  0D           Subroutine Parameter List:
DC      A(Qstring)       SQL Statement to execute
DC      A(LSQLDA)        Local SQLDA area
DC      A(Locca)         SQLCA returned from subroutine <----
DC      A(Hostvar)       Host Variable Workarea
DC      A(SaveRA)        Resource Adapter register savearea
:

```

In the Reentrant program, change from just referencing the "SQLCODE" variable, through the "LSQLCODE DSECT", to referencing the full "SQLCA" structure, through the "PASSEDCA DSECT", as follows:

```

:
L      R5,8(0,R1)        Point to Locca (SQLCA) return codes
USING PASSEDCA,R5       Reference Passed SQLCA structure
:
:
PASSEDCA DSECT ,         Passed SQLCA structure
EXEC SQL INCLUDE SQLCA  include SQLCA field definitions
:

```

Figure 94. SQLCA Changes for Driver/Reentrant Programs

Using Stored Procedures

The following example shows how to define the parameters in a stored procedure that uses the GENERAL linkage convention. PLIST=OS must be specified.

```

*****
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES *
* THE GENERAL LINKAGE CONVENTION. *
*****
A      CEEENTRY AUTO=PROG SIZE,MAIN=YES,PLIST=OS
      USING PROGAREA,R13
      .
      .
*****
* GET THE PASSED PARAMETER VALUES. THE GENERAL LINKAGE CONVENTION*
* FOLLOWS THE STANDARD ASSEMBLER LINKAGE CONVENTION: *
* ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS TO THE *
* PARAMETERS. *
*****
      L      R7,0(R1)          GET POINTER TO V1
      MVC   LOC V1(4),0(R7)    MOVE VALUE INTO LOCAL COPY OF V1

      .
      .
      L      R7,4(R1)          GET POINTER TO V2
      MVC   0(9,R7),LOC V2     MOVE A VALUE INTO OUTPUT VAR V2

      .
      .
      CEE TERM RC=0
*****
* VARIABLE DECLARATIONS AND EQUATES *
*****
R1      EQU   1                REGISTER 1
R7      EQU   7                REGISTER 7
PPA     CEEPPA ,              CONSTANTS DESCRIBING CODE BLOCK
      LTORG ,                  PLACE LITERAL POOL HERE
PROGAREA DSECT
      ORG   **CEEDSASZ         LEAVE SPACE FOR DSA FIXED PART
LOCV1   DS    F                LOCAL COPY OF PARAMETER V1
LOCV2   DS    CL9             LOCAL COPY OF PARAMETER V2

      .
      .
      .
PROG SIZE EQU   *-PROGAREA
      CEEDSA ,                MAPPING OF THE DYNAMIC SAVE AREA
      CEECAA ,                MAPPING OF THE COMMON ANCHOR AREA
      END   A

```

Figure 95. Stored Procedures - Using GENERAL Linkage Convention

The following example shows how to define the parameters in a stored procedure that uses the GENERAL WITH NULLS linkage convention.

```

*****
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES *
* THE GENERAL WITH NULLS LINKAGE CONVENTION *
*****
B      CEEENTRY AUTO=PROG SIZE,MAIN=YES,PLIST=OS
      USING PROGAREA,R13
*****
      .
      .
*****
* GET THE PASSED PARAMETER VALUES. THE GENERAL WITH NULLS LINKAGE*
* CONVENTION IS AS FOLLOWS: *
* ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N *
* PARAMETERS ARE PASSED, THERE ARE N+1 POINTERS. THE FIRST *
* N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS *
* WITH THE GENERAL LINKAGE CONVENTION. THE N+1ST POINTER IS *
* THE ADDRESS OF A LIST CONTAINING THE N INDICATOR VARIABLE *
* VALUES. *
*****
      L      R7,0(R1)          GET POINTER TO V1
      MVC    LOC V1(4),0(R7)   MOVE VALUE INTO LOCAL COPY OF V1
      L      R7,8(R1)          GET POINTER TO INDICATOR ARRAY
      MVC    LOCIND(2*2),0(R7) MOVE VALUES INTO LOCAL STORAGE
      LH     R7,LOCIND         GET INDICATOR VARIABLE FOR V1
      LTR    R7,R7            CHECK IF IT IS NEGATIVE
      BM     NULLIN          IF SO, V1 IS NULL

      .
      .
      .
      L      R7,4(R1)          GET POINTER TO V2
      MVC    0(9,R7),LOCV2     MOVE A VALUE INTO OUTPUT VAR V2
      L      R7,8(R1)          GET POINTER TO INDICATOR ARRAY
      MVC    2(2,R7),=H(0)     MOVE ZERO TO V2'S INDICATOR VAR

      .
      .
      .
      CEETERM RC=0
*****
* VARIABLE DECLARATIONS AND EQUATES *
*****
R1     EQU 1          REGISTER 1
R7     EQU 7          REGISTER 7
PPA    CEEPPA ,      CONSTANTS DESCRIBING THE CODE BLOCK
      LTRG ,         PLACE LITERAL POOL HERE
PROGAREA DSECT
      ORG  ++CEEDSASZ LEAVE SPACE FOR DSA FIXED PART
LOCV1  DS  F          LOCAL COPY OF PARAMETER V1
LOCV2  DS  CL9        LOCAL COPY OF PARAMETER V2
LOCIND DS  2H         LOCAL COPY OF INDICATOR ARRAY

      .
      .
      .
PROG SIZE EQU  *-PROGAREA
      CEEDSA ,        MAPPING OF THE DYNAMIC SAVE AREA
      CEECAA ,        MAPPING OF THE COMMON ANCHOR AREA
      END  B

```

Figure 96. Stored Procedure - Using GENERAL WITH NULLS Linkage Convention

Appendix B. Using SQL in C

Contents

Using ARIS6CD, a C Sample Program	270
Rules for Using SQL in C	270
Placing and Continuing SQL Statements	271
Delimiting SQL Statements	271
Identifying Rules for Case	271
Identifying Rules for Character Constants	272
Using the INCLUDE Statement	272
Using the CONNECT Statement	272
Using the C Compiler Preprocessor	272
Declaring Host Variables	272
Using Host Variables in SQL Statements	278
Using the Pointer Type Attribute	278
Using Host Variables as Function Parameters	280
Using C Variables in SQL: Data Conversion Considerations	280
Using C NUL-Terminated Strings and Truncation	281
Calculating Dates	281
Using Trigraphs	282
Using DBCS Characters in C	282
Considering Preprocessor-Generated Statements	282
Handling SQL Errors	285
Using Dynamic SQL Statements in C	286
Defining DB2 Server for VSE Data Types for C	287
Using Reentrant C Programs	289
Using Stored Procedures	289

Using ARIS6CD, a C Sample Program

ARIS6CD is a C language sample program for VSE systems that is shipped with the DB2 Server for VSE product. It resides on the production disk for the base product. You may find it useful to print this sample program before going through this appendix as the hard copy will provide an illustration for many of the topics discussed here.

The program satisfies the requirements of the application prolog and epilog. Near the beginning of the program all the host variables are declared, and error handling is defined. Near the logical end of the program, the database changes are rolled back, to assure that the database remains consistent for each use of the sample program. (For your own applications, of course, you will enter a COMMIT statement.)

To determine the types of C host variables to declare, refer to Figure 101 on page 287 which gives the C representation for each of the DB2 Server for VSE data types. Note the following:

- C expects a character array to end in a hex 00 null character when used to contain a character string. This null character is referred to as NUL, and is coded as '\0' in a C program. An SQL character string does not end in a NUL. Therefore, SQL will try to add a NUL to a character string when storing it in a character array host variable, and it will expect a NUL, which it will remove, when setting an SQL column value from a C character host variable. To account for the NUL, declare character array host variables to one character longer than the length of the SQL character data they are to contain. For all the rules concerning NULs, see "Using C NUL-Terminated Strings and Truncation" on page 281.

There are two other types of nulls to be aware of. Quite separate from the NUL character described above, C refers to a pointer value as NULL, in a similar way to PL/I. A C NULL pointer has a value of 0, and C allows the word NULL in pointer assignments and expressions. This use of NULL is distinct from the DB2 Server for VSE NULL, which means an undefined column or expression value. The word NULL can be used in a C program to mean either. You can always determine which is meant by the context.

When you are coding your own applications, you will need to obtain the data types of the columns that your host variables interact with. This can be done by querying the catalog tables. (These tables are described in the *DB2 Server for VSE & VM SQL Reference* manual.)

Rules for Using SQL in C

Placing and Continuing SQL Statements

All statements in your C program, including SQL statements, must be contained in columns 1 through 72 of your source file. Columns 73 through 80 can also be used if the NOSEquence C preprocessor option is specified; if NOSEquence is not specified, or if SEquence is specified, these columns will be ignored by the preprocessor. If NOSEquence is used, the NOSEQ and MARGINS(1,80) C compiler options must be used to compile the application program.

In a VSE environment, C comments may not start at column 1 because a “/*” starting at column 1 will be mistaken as an End-Of-Data-File JCL command. Other VSE restrictions may apply to the use of column 1 in C application programs.

Continuation of SQL statements and host variable declare statements across lines can be accomplished by breaking the line anywhere a blank can occur. Continuation of tokens (the basic syntactical units of a language) is allowed from one line to the next, by coding a backslash (the C continuation character with hex value X'E0') in the line to be continued immediately after the first part of the token (leaving the remainder of the line blank), and coding the next part of the token from column 1 on the continuation line. If column 1 on the continuation line is blank, the token is not continued. See the *DB2 Server for VSE & VM SQL Reference* manual for a discussion of tokens.

You can also use the trigraph ??/ in place of the backslash as the continuation character. The C preprocessor treats end-of-line like a blank delimiter except when it is in a literal.

Delimiting SQL Statements

Use delimiters on all SQL statements to distinguish them from regular C statements. You must begin each SQL statement in your program with EXEC SQL, and end each statement with a semicolon. EXEC and SQL must be in uppercase on the same line, with only blanks separating them (no in-line C or SQL comments). Also, EXEC SQL must be immediately followed by a blank, C comment, or SQL comment, and it must be preceded by either a blank, C comment, {, }, trigraph ??<, trigraph ??>,), colon, or semicolon.

Elsewhere within SQL statements, C and SQL comments are allowed anywhere that blanks are allowed. However, there must not be any comments within SQL statements that are dynamically defined and executed.

Any SQL statement except INCLUDE can be followed on the same line by another SQL statement, C statement, or C comment.

Identifying Rules for Case

The keywords EXEC SQL must appear in uppercase in your C program. The rest of an SQL statement can appear in mixed case, but will be interpreted as uppercase, except for host variable names and text within quotation marks, which will be left in the original case.

Note: C host variables are always treated with case sensitivity by the C preprocessor. This is true for the C compiler too, except for externals, which C may truncate and fold to uppercase. Keep this in mind when using host variables with external scope.

Identifying Rules for Character Constants

Remember to follow SQL, not C, conventions when coding such character constant strings. These strings must be delimited by single quotation marks, and an embedded backslash is not recognized as an escape character.

Using the INCLUDE Statement

To include external secondary input, specify:

```
EXEC SQL INCLUDE text_name
```

at the point in the source code where the secondary input is to be included. The *text_name* is the member name of a "B" type source member of a VSE library.

Use the SQL INCLUDE statement instead of the C preprocessor #include directive to include files that contain SQL host variables or SQL statements.

Using the CONNECT Statement

The CONNECT statement is required to establish a connection between the database manager and the program. To do an explicit connect, specify:

```
EXEC SQL CONNECT :userid IDENTIFIED BY :password;
```

Both *userid* and *password* must be host variables declared as fixed length 8 character strings.

Using the C Compiler Preprocessor

The preprocessor must run before the C compiler and its built-in preprocessor. It is therefore not possible to contain any C preprocessor directives within an SQL statement. The SQL INCLUDE statement should be used instead of the C #include for files that contain SQL host variable declarations or statements.

Declaring Host Variables

You must declare all host variables in an SQL declare section. For a description of an SQL declare section, refer to "Declaring Variables That Interact with the Database Manager" on page 6.

Declare host variables in the source file before the first use of the variable in an SQL statement. You can use the following types of variables in an SQL statement:

- Scalar variables
- Structure variables
- Structure elements
- Array variables

Scalar variables, structure elements, and array elements are **data objects**. For information on the use of these variables in an SQL statement, refer to "Using Host Variables" on page 54 and "Using Host Structures" on page 55.

Note: You can declare non-host variables in an SQL declare section; however, declarations that do not conform to DB2 Server for VSE declaration rules may return errors.

The definition of a host variable is subject to the following rules:

- A data object declared as a scalar variable or structure element may have any one of the following basic C data types:

short Short integer
long Long integer
float Floating-point
double Double-precision floating-point
decimal Decimal

The keyword **int** is optional in the declaration of a short or long integer. You cannot use the unqualified type **int** when declaring a variable to be used in an SQL statement: specify **short** or **long**.

- A data object declared as an array element can have any of the following basic C data types:

short Short integer
char Single character

- You can use scalar variables and structure elements as main variables. If they are declared with a data type of short integer, you can also use scalar variables as indicator variables.

- Character arrays hold the SQL CHARACTER data types. You should declare character arrays with one extra character to contain the string terminating NUL.

You can use the unsigned qualifier with character array host variables. It does not affect the way the system treats them.

An explicit constant decimal array size is required between the brackets, even if an initializer is used on the declare. Expressions, preprocessor functions (such as `sizeof`), octal or hex values, and `#defined` variables cannot be used as the size of character arrays.

`char value1 [5] = "TEST";`
or

Correct

`char value1 [5];`

Incorrect

`char value1 [] = "TEST";`

Incorrect

`char value1 [sizeof(var)];`

Incorrect

`char value1 [MAXLEN + 1]`

Incorrect
(octal)

`char value1 [015];`

- You can only use short integer arrays as indicator arrays. The following is an example of an indicator array:

```
short ind_array [10];
```

You cannot use indicator array elements as main or indicator variables.

- You can use a structure variable as a host structure or as a varying-length string definition. The structure declaration must be in the following form when used to define a varying-length string:

```
struct tag {
    short vlen;
    char vstr[nnn];
} varname1;
```

The structure tag is optional. Any legal C names can be used for the structure and the contained variables. The nnn, defining the length of the largest string to be held in the structure, is specified by you.

This structure defines a VARCHAR or LONG VARCHAR host variable with the name varname1 and a length nnn. You cannot use this structure as a host structure; you cannot use the elements of the structure as host variables.

The system does not add or expect a NUL at the end of a VARCHAR or LONG VARCHAR string. If one is needed, you can use a character array host variable, or you can add one after the data value has been returned, with the statement:

```
varname1.vstr[varname1.vlen]='\0';
```

If a NUL is required, ensure that the nnn is one larger than the maximum allowable string, so that adding the NUL at the end will never overflow the allocated storage.

A macro is provided to assist in the declaration of VARCHAR structures:

```
SQLVARCHAR(varname,nnn)
```

will expand to:

```
struct{
    short sqlen;
    char sqlstr[nnn];
}varname;
```

You can use this macro wherever a structure declaration defines a varying-length string.

- A structure variable which defines a host structure is any two-level structure, other than a varying-length string definition, declared in an SQL declare section. The following example is a host structure:

```
struct tag{char projno [7];
    short actno;
    long acstaff;
    char acstdate [10];
    char acendate [10];
}projstrct;
```

Note: The structure tag is optional.

This structure represents the following list of host variables when used in an SQL statement:

projno, actno, acstaff, acstdate, acendate

In other words, the two following SQL statements are equivalent:

```
EXEC SQL SELECT PROJNO, ACTNO, ACTSTAFF, ACSTDATE, ACENDATE
        INTO :projstruct
        FROM PROJ_ACT
        WHERE PROJNO = '100000'
```

```
EXEC SQL SELECT PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
        INTO :projno, :actno, :acstaff, :acstdate, :acendate
        FROM PROJ_ACT
        WHERE PROJNO = '100000'
```

A host structure can either be a stand-alone structure or a substructure of a more complex structure. The following example is a complex structure that contains a host structure:

```
struct tag { char empno [7];
            struct taga { char firstname [13];
                        char midinit [1];
                        char lastname [16];
                        } empname;
            char workdept [3];
            char phoneno [4];
        } employee;
```

The structure empname is a host structure.

You can use the elements of the host structure and the elements of a complex structure containing a host structure as host variables. In the previous example, you can use empno, firstname, midinit, lastname, workdept and phoneno as host variables.

You can code a substructure in the host structure to represent a varying-length string element if the substructure conforms to the rules for a varying-length string definition. All of the rules in the description of structures that define varying-length strings also apply in this situation. The following example is a host structure that contains a VARCHAR element:

```
struct tag { struct taga { short fnlen;
                        char fntext [12];
                        } firstname;
            char midinit [1];
            struct tagb {short lnlen;
                        char lntext [15];
                        } lastname;
        } empname;
```

The C preprocessor interprets the structure empname as a host structure containing 3 elements: firstname with data type VARCHAR and length 12, midinit with data type CHAR and length 1, and lastname with data type VARCHAR, and length 15.

Note: Any structure matching the description of a varying-length string definition is interpreted as a VARCHAR or LONG VARCHAR variable and cannot be used as a host structure.

- Third level host structures are permitted in C to support varying-length strings. The following is an example of varying-length string declarations in host structures.

```

EXEC SQL BEGIN DECLARE SECTION;
struct
{
    char  last??(9??);
    char  first??(9??);
    struct
    {
        short  addlen;
        char  addtext??(200??);
    } address;
} empname;
EXEC SQL END DECLARE SECTION;

main()
{

    EXEC SQL SELECT LASTNAME, FIRSTNME, ADDRESS
        INTO :empname
        FROM EMPLOYEE
        WHERE LASTNAME = 'JOHANSON';

}

```

In this example, empname is considered by the C preprocessor to be a two-level structure because the structure of address matches that of a VARCHAR data type. As a result, empname may be used in the SELECT statement. If, for example, addlen was changed from short to long, the structure of address would no longer match a VARCHAR data type and empname would be considered a three-level structure. As a result, empname could NOT be used in a SELECT statement.

- Union, enumeration, bitfield, and void types are not supported. Typedefs are not supported.
- Auto, static, extern, const, volatile, and _Packed storage classes are supported. The register storage class is not supported. If no class is specified, the usual C default storage class applies (which depends on the placement of the declaration within the source file).
- The system supports any sequence of declaration keywords that is also supported by the C compiler.
- Initialization of variables on the declaration statement is supported.
- You can declare multiple variables of the same type in the same C declaration statement. For example:

```

static short partno, suppno, time;
static char name [16] , adr[36];
static double qonhand, qonorder, price;

```

Note: You must explicitly declare all structures in C within the multi-level structure. You cannot declare a structure for reference within another structure (except for the SQLVARCHAR macro). In the following example, only dates and product may be used as host structures. orderno and custnum may be used as scalar host variables and may be qualified as custord.orderno and ordinfo.custnum or custord.ordinfo.custnum.

```

EXEC SQL BEGIN DECLARE SECTION;
struct
{
    char  orderno??(10??);
    struct
    {
        char custnum??(10??);
        struct
        {
            char ordate??(7??);
            char delivdte??(7??);
        } dates;
    } ordinfo;
} product ;
} custord ;
EXEC SQL END DECLARE SECTION;

main()
{

    EXEC SQL SELECT STOCKNO, QUANTITY
        INTO :product
        FROM ORDER
        WHERE STOCKNO = '1234567890';

}

```

- You cannot duplicate variable names in a single source file, even if they are in different blocks or functions. The C preprocessor defines a duplicate variable name as any name that cannot be referenced unambiguously when fully qualified. (After a variable is declared in an SQL declare section, it is known to SQL for all functions and blocks for the rest of the source file, regardless of the host variable's actual scope. Therefore, that variable, or another variable with the same name and type, cannot be used in an SQL statement, even in a scope outside of the original SQL declare section.) See “Using Host Variables as Function Parameters” on page 280 for more information.
- The database manager allows host variable names, statement labels, and SQL descriptor area names of up to 256 characters in length, subject to any C language restrictions mentioned in this appendix.
 - Note:** Because of the restriction on the number of host variables in a statement, host structures with greater than 256 fields will not be allowed.
- You should not declare variables whose names begin with SQL, sql, RDI, or rdi unless otherwise instructed. These names are reserved for the database manager use.
- Host variable names are case-sensitive. For example, a host variable called partno is different from one called PartNo.
- Rules for continuation are the same as those described for SQL statements.

Note that other program variables can also be declared as usual outside the SQL declare section. The previous restrictions do not apply to non-SQL declarations.

Using Host Variables in SQL Statements

When you reference host variables, host structures, structure fields or indicator arrays in an SQL statement, you must precede each reference by a colon (:). The colon distinguishes these variables from SQL identifiers (such as column names). The colon is not required outside an SQL statement.

Using the Pointer Type Attribute

Scalar host variables can be defined as pointers to any C data type that the database manager supports. The following rules and restrictions apply:

- For the basic C data types, the variable must be declared in the same way it is referenced in an SQL statement. For example:

```
short *partno;  
...  
SELECT PARTNO INTO :*partno ...
```

- In the case of character arrays, the array size must be explicitly defined in the declaration. For example:

```
char (*v1_ptr) [5];
```

Correct

```
char (*v1_ptr) [ ];
```

Incorrect

```
char *v1_ptr ;
```

Incorrect

The use of parentheses is required in order for arrays to define a pointer to an array of 5 characters, as opposed to an array of 5 pointers to characters. (See “Using C NUL-Terminated Strings and Truncation” on page 281 for the limitations on string lengths.)

The host variable would then be referenced as *v1_ptr in an SQL statement.

- The asterisk is considered part of the host variable name. This means:
 - The asterisk is included in the 256-character length limitation for host variable names.
 - If a host variable is declared with an asterisk, it must always be used within the SQL statement with the asterisk. If it was declared without an asterisk, then it must never have one in an SQL statement.
- The programmer is responsible for ensuring that the pointer is set before it is used.

There are primarily two uses for pointer types with SQL statements:

1. Allocating or sharing storage.

A program could contain a single SQL declare section, and use pointers for some or all large data areas. Then, before a pointer is used in an SQL statement, an alloc function could be used to acquire storage and set the

pointer, or the pointer could be set to a shared storage area. This allows the program to reduce its overall storage requirement. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    struct tag {
        short vlen;
        char vstr[1000];
    } *vstr_ptr;
    ...
EXEC SQL END DECLARE SECTION;

...
vstr_ptr = (struct tag *) malloc(sizeof(struct tag));

EXEC SQL SELECT DESCRIPTION
        INTO :*vstr_ptr FROM TABLE;

...
}
```

2. Passing variables to functions for update.

C usually passes parameters by value. This prevents the called function from changing the caller's version of a parameter. Passing a parameter by reference can be accomplished by the caller explicitly passing a pointer to the data. The called function then changes the data referenced by the pointer by using the asterisk for indirection. If the value is to be changed with an SQL statement, the called function must also declare the pointer value of the parameter in an SQL DECLARE section. For example:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
    ...
    long int partno;
    ...
EXEC SQL END DECLARE SECTION;

    getdata(&partno);
    ...
}

getdata(partno_ptr)
EXEC SQL BEGIN DECLARE SECTION;
long int *partno_ptr;
EXEC SQL END DECLARE SECTION;
{
    EXEC SQL SELECT PARTNO
        INTO :*partno_ptr FROM TABLE;

    ...
}
```

Using Host Variables as Function Parameters

Host variables with the same name can only be declared in one SQL declare section. When passing a host variable to a function within the same file as the calling function, the variable can be used in an SQL statement in the called function without being redeclared in an SQL declare section. For example:

```
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        ...
        long int partno;
        ...
    EXEC SQL END DECLARE SECTION;

    getdata(partno);
    ...
}

getdata(partno)
long int partno;
{
    EXEC SQL BEGIN DECLARE SECTION;
        long int qonhand;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT QONHAND
        INTO :qonhand FROM TABLE
        WHERE PARTNO = :partno;
    ...
}
```

Given the SQL declaration of `partno` within `main()`, `partno` can be used in any SQL statement that follows it in the file. If `getdata` had a different name for the `partno` parameter, it would have to be included in `getdata`'s SQL declare section.

For information on how to allow a called function to update a parameter, refer to "Using the Pointer Type Attribute" on page 278.

Using C Variables in SQL: Data Conversion Considerations

Host variables must be type-compatible with the columns with which they are to be used. For example, if you want to retrieve into a program variable the `QONHAND` column of the database, and the data type of `QONHAND` is `INTEGER`, you should declare the program variable to be of type `short`, `long`, `float`, or `double`.

The database manager considers the numeric data types compatible as well as the character string data types (`CHAR`, `VARCHAR`, and `LONG VARCHAR`, including strings of different declared lengths). Of course, an overflow condition may result if, for example, you assign a 31-bit integer to a 15-bit integer and the current value of the 31-bit integer is too large to fit in 15 bits. Truncation also occurs when a decimal number having a scale greater than zero is assigned to an integer. In general, overflow occurs when significant digits are lost, and truncation occurs when nonsignificant digits are lost.

The system also considers the datetime data types to be compatible with character data types (`CHAR`, and `VARCHAR`, but not long fields).

Using C NUL-Terminated Strings and Truncation

The database manager interprets a character string in C as NUL-terminated if the length of the string is greater than 1 byte and less than 32,768 bytes.

The NUL-byte is mandatory when the database manager receives data from a NUL-terminated string. You receive an SQLCODE -302 (SQLSTATE '22024') if the NUL-byte is not found within the defined length of the string. This means that the maximum number of bytes of data that can be stored in a NUL-terminated string is one less than the defined length of the string.

When data is sent from the application server to a NUL-terminated string, a NUL-byte is always appended to the end of the string. If the variable is not big enough to hold the entire string (including the NUL), then a warning condition is indicated using the SQLCA SQLWARN flags and the output indicator value, as shown in the following chart. Truncation will occur even in the case where a character value of actual length n is to be assigned to a C variable declared as length n due to the NUL character being inserted at the last byte of the declared length. When truncation occurs, that last byte is overwritten by NUL.

Figure 97. Warning Flags after Character Truncation

Condition	SQLCA SQLWARN0	SQLCA SQLWARN1	Output Indicator Variable (if supplied)
Character string, including the NUL, fits in the declared C character array.	blank	blank	0 (zero)
Actual data truncated. That is, the C variable declared as less than or equal to n , to hold a character value of actual length n .	W	W	Original length of value (n) excluding the NUL.

Calculating Dates

Date calculations can result in date durations, and the database manager converts the result into any numeric type of a column or a host variable. However, to involve a date duration in a calculation (for example, to add a duration to a date), the date duration must be in DECIMAL(8,0) format. The system does not automatically convert any numeric type of column or host variable to a decimal value for use in a date calculation. If your C compiler does not support the fixed decimal data format, the scalar "DECIMAL" conversion function must be used to explicitly convert a value to decimal type. For example,

```
long duration=10100; /* 1 year and 1 month */
long result_dt;
```

```
EXEC SQL SELECT START_DATE+DECIMAL(:duration,8,0)
          INTO :result_dt FROM TABLE;
```

Using Trigraphs

A trigraph is a sequence of three characters that you write in place of a C source character that your input device does not generate. The following trigraphs are supported by the C preprocessor in an SQL declare section:

```
??(    [ (left bracket)
??)    ] (right bracket)
??<   { (left brace)
??>   } (right brace)
```

The following trigraph is supported in an SQL statement only when used as a continuation character:

```
??/    \ (backslash)
```

Using DBCS Characters in C

The rules for the format and use of DBCS characters in SQL statements are the same for C as for other host languages supported by the system. For a discussion of these rules, see “Using a Double-Byte Character Set (DBCS)” on page 50.

The C language does not provide a way to define graphic host variables. If you want to add graphic data to or retrieve it from DB2 Server for VSE tables, you must execute the affected statements dynamically. By doing so, the data areas that are referenced by each statement can be described in an SQLDA. In the SQLDA, you must set the data type of the areas containing graphic data to one of the graphic data types. For a discussion of the SQLDA, refer to the *DB2 Server for VSE & VM SQL Reference* manual.

Considering Preprocessor-Generated Statements

When preprocessing an SQL C program, every executable SQL statement is translated into control block declarations, assignment statements, and a function call to pass the control block to the preprocessor at run time. To simplify the generation of this code during preprocessing, a number of typedef and communication area definitions are placed just after any initial C compiler directives or comments.

In addition, to assist the application programmer, the SQLVARCHAR macro is inserted with the typedefs and communication area definitions.

The preprocessor-generated statements are described in Figure 98 on page 282. These statements are inserted immediately before the first line in the source program that is not a blank line, a C comment, or a C precompiler directive.

The C preprocessor imposes two restrictions on the coding of C precompiler directives:

1. You may not use the #INCLUDE precompile directive to include the main function of a C program.
2. Conditional precompiler directives that contain C code must come after the first non-precompiler directive.

Figure 98 (Page 1 of 3). C Preprocessor-Generated Statements

Generated Code	Purpose
#pragma linkage (ARIPRDI,0S)	To establish correct addressability and parameter passing conventions with the system at run time.
<pre>#ifndef SQLVARCHAR #define SQLVARCHAR(varname,nnn) \ struct { \ short sqllen; \ char sqlstr[nnn]; \ } varname #endif</pre>	Macro that can be used by the application to simplify the C program. The definition of this macro can be changed by including a #define statement before the first non-compiler directive C statement or SQL statement in your program.

Figure 98 (Page 2 of 3). C Preprocessor-Generated Statements

Generated Code	Purpose
<pre> typedef struct { short CALLTYPE; char AUTHOR[8]; short PROG_NAME1; char PROG_NAME[8]; short SECTION_NUM; short CLASS_SECTION; char *CODEPTR; char *VPARAMPTR; char *AUXPARAMPTR; char *SQLTIEPTR; char SPECIALCALL; char CALLFLAG; char WAITFLAG; char RELEASEFLAG; char VPARAMIND; char AUXPARAMIND; char ERRORFLAG; char RDIDESCFLAG; long MAILBOXLEN; char RDIRELNO; char RDICISL; char RDIDATE; char RDITIME; long RDIFDBCK; char *RDIEXTP; char RDIRESV1[2]; char RDIRDB16; char RDIRESV2; } SQL_RDIIN; typedef struct { char *RDIPTR01; char *RDIPTR02; } SQL_RDIPT; typedef struct { short DATA_TYPE; short LEN; char *DATA_PTR; short *INFOPTR; short NAME1; char NAME[30]; } SQL_PVELMS; typedef SQL_PVELMS *SQL_PVLM; typedef struct { short CURSRLEN; char CURSRNAM[18]; } SQL_RDICURAR; typedef union { long rdicnst1[2]; char rdicnstc[8]; } SQL_RDICONST; </pre>	<p>typedefs of allocated control blocks used when translating executable SQL statements into C function calls.</p>

Figure 98 (Page 3 of 3). C Preprocessor-Generated Statements

Generated Code	Purpose
<pre>static long SQLTIE[12]; static char RDIRDBN[16]; static struct { char RDIEXTEC[8]; long RDIEXTFLR; char *RDIDBNMP; char *RDICONSP; char *RDIBPOPT; char *RDIXPTRS[6]; } SQLRDIX;</pre>	Communication areas used to save information about the state of the C program between run-time calls to the database manager.

Handling SQL Errors

A return code structure (the SQLCA) must be in scope for each executable SQL statement. You can define one by coding the following statement in your source program:

```
EXEC SQL INCLUDE SQLCA;
```

The preprocessor replaces this statement with the declaration of the SQLCA structure, and a set of #defines to make referring to the error codes and flags easier. These are shown in Figure 99.

```
#ifndef SQLCODE
struct sqlca
{
    unsigned char sqlcaid[8];
    long sqlcabc;
    long sqlcode;
    short sqlerrml;
    unsigned char sqlerrmc[70];
    unsigned char sqlerrp[8];
    long sqlerrd[6];
    unsigned char sqlwarn[11];
    unsigned char sqlstate[5];
};
#define SQLCODE sqlca.sqlcode
#define SQLWARN0 sqlca.sqlwarn[0]
#define SQLWARN1 sqlca.sqlwarn[1]
#define SQLWARN2 sqlca.sqlwarn[2]
#define SQLWARN3 sqlca.sqlwarn[3]
#define SQLWARN4 sqlca.sqlwarn[4]
#define SQLWARN5 sqlca.sqlwarn[5]
#define SQLWARN6 sqlca.sqlwarn[6]
#define SQLWARN7 sqlca.sqlwarn[7]
#define SQLWARN8 sqlca.sqlwarn[8]
#define SQLWARN9 sqlca.sqlwarn[9]
#define SQLWARNA sqlca.sqlwarn[10]
#define SQLSTATE sqlca.sqlstate
#endif
struct sqlca sqlca;
```

Figure 99. SQLCA Structure (in C)

Note: SQLCA character array variables are not NUL-terminated. They cannot be directly used by C string manipulation functions.

The SQLCA must not be declared within the SQL declare section. It may be declared outside all functions in the module, which gives it global scope, or separately within each function that contains executable SQL statements.

Instead of using the SQL INCLUDE SQLCA statement, the SQLCA can be coded directly, or #included from a header file.

You may find that the only variable in the SQLCA that you really need is SQLCODE. If this is the case, declare just the SQLCODE variable, and invoke NOSQLCA support at preprocessor time.

The number of SQLCODE declarations is not limited by the DB2 Server for VSE preprocessor. If a stand-alone SQLCODE is specified, the code inserted by the preprocessor into the C code to expand an EXEC SQL statement will refer to the address of that SQLCODE. The C compiler determines if multiple declarations within a program section are not acceptable. In addition, the C compiler determines which region of the code an SQLCODE declaration refers to.

Using Dynamic SQL Statements in C

You must declare an SQLDA structure to execute dynamically defined SQL statements. You can have the database manager include the structure definition automatically, by specifying the following statement in your source code:

```
EXEC SQL INCLUDE SQLDA;
```

You can also include the structure definition by directly coding it as shown in Figure 100.

```
#ifndef SQLDASIZE
struct sqlda {
    unsigned char sqldaid[8];
    long sqldabc;
    short sqln;
    short sqld;
    struct sqlvar {
        short sqltype;
        short sqlen;
        unsigned char *sqldata;
        short *sqlind;
        struct sqlname {
            short length;
            unsigned char data[30];
        } sqlname;
    } sqlvar[1];
};
#define SQLDASIZE(n) \
    (sizeof(struct sqlda)+((n)-1)* \
    sizeof(struct sqlvar))
#endif
```

Figure 100. SQLDA Structure (in C)

Note: The SQLDA character array variables `sqldaid` and `sqlname.data` are not NUL-terminated. They cannot be directly used by C string manipulation functions.

The SQLDA must not be declared within the SQL declare section.

Using the defined preprocessor function `SQLDASIZE`, your program can dynamically allocate an SQLDA of adequate size for use with each EXECUTE statement. For example, the code fragment below allocates an SQLDA that is adequate for five fields, and uses it in an EXECUTE of statement S3:

```
struct sqlda *daptr;

daptr = (struct sqlda *)malloc(SQLDASIZE(5));
daptr->sqln=5;
/* Add code to set the rest of values and
   pointers in the SQLDA */
EXEC SQL EXECUTE S3 USING DESCRIPTOR *daptr;
```

Note: The variable that points to the SQLDA is not defined in an SQL declare section. Its context within an SQL statement (following INTO or USING DESCRIPTOR) is enough to identify it.

You can use a similar technique to allocate an SQLDA for use with a DESCRIBE statement. The following program fragment illustrates the use of SQLDA with DESCRIBE for three fields and a “prepared” statement S1:

```
struct sqlda *daptr;

EXEC SQL DECLARE C1 CURSOR FOR S1;
daptr = (struct sqlda *)malloc(SQLDASIZE(3));
daptr->sqln=3;
EXEC SQL DESCRIBE S1 INTO *daptr;
if (daptr->sqld > daptr->sqln)
    --get a bigger one
Set sqldata and sqlind
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 USING DESCRIPTOR *daptr;
```

There is no standard C type to support packed decimal data. If you want to get data in packed decimal format, the SQLDA must be filled in with an `SQLTYPE` of 484 and with the appropriate values for precision and scale in `SQLEN`. The C program would then have to deal with the data in its packed format.

See the *DB2 Server for VSE & VM SQL Reference* manual for more information on the individual fields within SQLDA.

Defining DB2 Server for VSE Data Types for C

Figure 101 (Page 1 of 2). DB2 Server for VSE Data Types for C

Description	DB2 Server for VSE Keyword	Equivalent C Declaration
A binary integer of 31 bits, plus sign.	INTEGER or INT	long or long int
A binary integer of 15 bits, plus sign.	SMALLINT	short or short int
A packed decimal number, precision p , scale s ($1 \leq p \leq 31$ and $0 \leq s \leq p$). In storage, the number occupies a maximum of 16 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL[(p , s)] or DEC[(p , s)] 1	decimal(p , s) If your version of the C compiler does not provide support for the decimal data type, C short, long, float and double host variables are supported for conversion to and from DECIMAL columns. To preserve decimal places: if $p < 7$ use float; else use double.
A single-precision (4-byte) floating-point number, in short System/390 floating-point format.	REAL or FLOAT(p), $1 \leq p \leq 21$	FLOAT
A double-precision (8-byte) floating-point number, in long System/390 floating-point format.	FLOAT or FLOAT(p), $22 \leq p \leq 53$ or DOUBLE PRECISION	DOUBLE
A fixed-length character string of length 1.	CHARACTER[(1)] or CHAR[(1)]	char or char ..[1]
A NUL-terminated character string of maximum defined length n . Range of n is $1 \leq n \leq 254$. The terminating NUL is mandatory upon input.	VARCHAR(n)	char ..[$n+1$]
A NUL-terminated character string of maximum defined length of 32 767 bytes, subject to certain usage limitations. Range of n is $255 \leq n \leq 32\,766$. The terminating NUL is mandatory upon input.	LONG VARCHAR	char ..[$n+1$]
A varying-length character string of maximum length n . If $n > 254$ or $\leq 32\,767$, this data type is considered a long field. See "Using Long Strings" on page 45 for more information.	VARCHAR(n)	struct { short ..; char ..[n]; }
A varying-length character string of maximum length 32 767 bytes, subject to certain usage limitations.	LONG VARCHAR	struct { short ..; char ..[n]; }

Figure 101 (Page 2 of 2). DB2 Server for VSE Data Types for C

Description	DB2 Server for VSE Keyword	Equivalent C Declaration
A fixed-length string of n DBCS characters where $0 < n \leq 127$.	GRAPHIC[(n)]	Not supported
A varying-length string of n DBCS characters. If $n > 127$ or ≤ 16383 , this data type is considered a long field. See "Using Long Strings" on page 45 for more information.	VARGRAPHIC(n)	Not supported
A varying-length string of DBCS characters of maximum length 16383, subject to certain usage limitations.	LONG VARGRAPHIC	Not supported
A NUL-terminated or varying-length character string representing a date.	DATE	see VARCHAR(n)
A NUL-terminated or varying-length character string representing a time.	TIME	see VARCHAR(n)
A NUL-terminated or varying-length character string representing a timestamp.	TIMESTAMP	see VARCHAR(n)

Notes:

1. NUMERIC is a synonym for DECIMAL, and may be used when creating or altering tables. In such cases, however, the CREATE or ALTER function will establish the column (or columns) as DECIMAL.
2. For a NUL-terminated string, the declared length should be one more than the maximum length of a datetime to allow for the terminating NUL-byte, which is mandatory input. Refer to the *DB2 Server for VSE & VM SQL Reference* manual for information on minimum and maximum lengths.

Using Reentrant C Programs

A reentrant program has the characteristic of dynamic allocation of space for data and save areas. This characteristic can be employed in C programs. In this case, the data and save areas are dynamically allocated in a "static" area by the IBM C Program Product Compiler.

Using Stored Procedures

Figure 102 on page 290 shows how to define the parameters in a stored procedure that uses the GENERAL linkage convention.

- *argv* contains an array of pointers to the parameters that were passed to the stored procedure.
 - *argv*[0] is a special entry containing the address of the stored procedure name
 - *argv*[1] contains the address of parameter 1
 - *argv*[2] contains the address of parameter 2

- *argc* contains the number of parameters that were passed to the stored procedure, plus one to account for the procedure name which is passed in *argv[0]*.

```

#pragma options(RENT)
#pragma runopts(PLIST(OS))
#include <stdlib.h>
#include <stdio.h>
/*****
/* Code for a C language stored procedure that uses the      */
/* GENERAL linkage convention.                               */
*****/
main(argc,argv)
    int argc;                /* Number of parameters passed */
    char *argv[];           /* Array of strings containing */
                           /* the parameter values      */
{
    long int locv1;         /* Local copy of V1          */
    char locv2[10];        /* Local copy of V2          */
                           /* (null-terminated)        */

    .
    .
    .
/*****
/* Get the passed parameters.                               */
*****/
    if(argc==3)            /* Should get 3 parameters:  */
    {                       /* procname, V1, V2          */
        locv1 = *(int *) argv[1];
                           /* Get local copy of V1     */

        .
        .
        .
        strcpy(argv[2],locv2);
                           /* Assign a value to V2    */

        .
        .
        .
    }
}

```

Figure 102. Stored Procedure - Using GENERAL Linkage Convention

Figure 103 on page 291 shows how to define the parameters in a stored procedure that uses the GENERAL WITH NULLS linkage convention. In this case:

- *argv[0]* contains the address of the stored procedure name
- *argv[1]* contains the address of parameter 1
- *argv[2]* contains the address of parameter 2
- *argv[n]* contains the address of parameter *n*
- *argv[n+1]* contains the address of the indicator variable array

```

#pragma runopts(PLIST(OS))
#include <stdlib.h>
#include <stdio.h>
/*****
/* Code for a C language stored procedure that uses the
/* GENERAL WITH NULLS linkage convention.
*****/
main(argc,argv)
    int argc;          /* Number of parameters passed */
    char *argv[];     /* Array of strings containing
                      /* the parameter values
{
    long int locv1;    /* Local copy of V1
    char locv2[10];   /* Local copy of V2
                      /* (null-terminated)
    short int locind[2]; /* Local copy of indicator
                      /* variable array
    short int *tempint; /* Used for receiving the
                      /* indicator variable array

.
.
.
/*****
/* Get the passed parameters.
*****/
if(argc==4)          /* Should get 4 parameters:
{
    /* procname, V1, V2,
    /* indicator variable array

    locv1 = *(int *) argv[1];
    tempint = argv[3]; /* Get pointer to indicator
                      /* variable array
    locind[0] = *tempint;
                      /* Get 1st indicator variable
    locind[1] = *(++tempint);
                      /* Get 2nd indicator variable
    if(locind[0]<0) /* If 1st indicator variable
    {
        /* is negative, V1 is null

.
.
.
}

.
.
.
strcpy(argv[2],locv2);
                      /* Assign a value to V2
* (++tempint) = 0;    /* Assign 0 to V2's indicator
                      /* variable
}
}

```

Figure 103. Stored Procedure - Using GENERAL WITH NULLS Linkage Convention

Appendix C. Using SQL in COBOL

Contents

Using ARIS6CBD, a Sample COBOL Program	294
Rules for Using SQL in COBOL	294
Placing and Continuing SQL Statements	294
Delimiting SQL Statements	295
Identifying Rules for Case	296
Declaring Host Variables	296
Using Host Variables in SQL Statements	299
Using Long VARCHAR Host Variables	300
Using Preprocessor Options	300
Using the QUOTE Parameter	300
Using the COB2 Parameter	300
Invoking COPYBOOKs	300
Using the COBRC Parameter	300
Using the TRUNC Compiler Option	301
Using the INCLUDE Statement	301
Using COBOL Variables in SQL: Data Conversion Considerations	301
Other Coding Considerations	302
Using DBCS Characters in COBOL	302
Handling SQL Errors	302
Using Dynamic SQL Statements in COBOL	304
Defining DB2 Server for VSE Data Types for COBOL	305
Using Reentrant COBOL Programs	308
Using the DYNAM Compiler Option	308
Using Stored Procedures	308

Using ARIS6CBD, a Sample COBOL Program

ARIS6CBD is a COBOL language sample program for VSE systems that is shipped with the DB2 Server for VSE product. It resides on the production disk for the base product. You may find it useful to print this sample program before going through this appendix as the hard copy will provide an illustration for many of the topics discussed here.

Here is a summary of the program by COBOL Divisions:

- Identification and Environment Divisions

You do not have to do anything different in either of these divisions for DB2 Server for VSE applications.

- Data Division

In the Data Division of any COBOL application, you must declare all host variables and the SQLCA structure.

The only SQL statements allowed in the Data Division are those shown in the sample program and the INCLUDE statement; all others must be placed in the Procedure Division.

The COBOL PICTURE clauses for the host variables are determined by referring to Figure 106 on page 305 which gives the COBOL representation for each of the DB2 Server for VSE data types. When you are coding your own applications, you will need to obtain the data types of the columns that your host variables interact with. This can be done by querying the catalog tables, which are described in the *DB2 Server for VSE & VM SQL Reference* manual.

- Procedure Division

The program must explicitly connect to the application server. WHENEVER statements should be coded to provide for error handling. Near the logical end of the program, the database changes are rolled back, to ensure that the database remains consistent for each use of the sample program. (For your own applications, of course, you will enter a COMMIT statement.)

Rules for Using SQL in COBOL

In this appendix, the term COBOL implies OS/VS COBOL, VS COBOL II, or VSE IBM COBOL for VSE.

Placing and Continuing SQL Statements

Figure 104 shows how SQL statements can be coded

Figure 104 (Page 1 of 2). Coding SQL Statements in COBOL Program Sections

SQL Statement	Program Section
BEGIN DECLARE SECTION	WORKING STORAGE or
END DECLARE SECTION	LINKAGE SECTION or
	FILE SECTION

Figure 104 (Page 2 of 2). Coding SQL Statements in COBOL Program Sections

SQL Statement	Program Section
INCLUDE SQLCA	WORKING-STORAGE SECTION
INCLUDE <i>text_file_name</i>	PROCEDURE DIVISION or DATA DIVISION
Other	PROCEDURE DIVISION SQL statements are coded between columns 12 and 72 inclusive.

The system checks that SQL statements are not used in nested programs. Also if one program immediately follows another program, the second program must not contain SQL statements.

The rules for continuation of tokens from one line to the next are the same as the COBOL rules for the continuation of words and constants. If a string-constant is continued from one line to the next, the first non-blank character in that next line must be a single quotation mark (') or a double quotation mark ("). If a delimited SQL identifier (such as "EMP TABLE") is continued from one line to the next, the first non-blank character in that next line must be a double quotation mark. COBOL comment lines, identified by an asterisk * in column 7, can be coded within an embedded statement.

Delimiting SQL Statements

Delimiters are required on all SQL statements to distinguish them from regular COBOL statements. You must precede each SQL statement with EXEC SQL, and terminate each one with END-EXEC. Any desired COBOL punctuation, such as a period, can be placed after the END-EXEC. For example, suppose an SQL statement occurs as one of several statements nested inside a COBOL IF-statement. In this instance, the SQL statement should not be followed by a period.

EXEC SQL must be specified within one line; the same is true for END-EXEC. A separator (such as a blank space, SQL comment, or end-of-line) must precede the END-EXEC that terminates an SQL statement; however, no punctuation is required after the END-EXEC.

If an SQL statement appears within an IF sentence such that a COBOL ELSE clause immediately follows the SQL statement, the clause must begin with the word ELSE. In addition, this ELSE must be contained entirely on one line. (No continuation is allowed for the word ELSE).

SQL WHENEVER and DECLARE CURSOR statements should *not* be the only contents of COBOL IF or ELSE clauses as the preprocessor does not generate COBOL code for these statements.

If an SQL statement terminates a COBOL IF sentence, a period should immediately follow END-EXEC with no intervening blanks. A blank should follow the period.

Because a COBOL statement can be immediately preceded by a paragraph name, so can an embedded SQL statement. Similarly, an embedded SQL statement in the Procedure Division can be immediately followed by a separator period.

Identifying Rules for Case

Mixed case can be used in your COBOL program. The SQL preprocessor will change the lowercase into uppercase, except for text within quotation marks, which will be left in the original case.

Declaring Host Variables

You must declare all host variables in an SQL declare section. For a description of an SQL declare section, refer to “Declaring Variables That Interact with the Database Manager” on page 6.

Declare host variables in the source file before the first use of the variable in an SQL statement. All SQL declare sections must be located in the Working-Storage Section, the File Section, or the Linkage Section of the Data Division. You can use the following types of variables in an SQL statement:

- Elementary items (independent or subordinate of a group item)
- Group items
- Tables

For information on the use of these variables in an SQL statement, refer to “Using Host Variables” on page 54 and “Using Host Structures” on page 55.

Note: You can declare non-host variables in an SQL declare section; however, declarations that do not conform to DB2 Server for VSE declaration rules may return errors.

The declaration of a host variable is subject to the following rules:

- All elementary items that are declared in an SQL declare section can be used as main variables. If these items are declared with a data type of short integer, they can also be used as indicator variables.
- The only tables accepted by the COBOL preprocessor are tables of short integer elements. These may only be used as indicator arrays. The following example is an indicator array declaration:

```
01 IND_ARRAY.  
   05 IND-ELEMENT OCCURS 15 TIMES PIC S9(4) COMP.
```

The COBOL preprocessor recognizes IND-ELEMENT as the indicator array.

You cannot use indicator array elements as main variables or indicator variables.

- You can use a group item as a host structure or as a varying-length string definition. The structure must take the following form when used to define a varying-length string:

```
01 VARCHAR-FIELD.  
   49 LEN-FIELD PIC S9(4) COMP.  
   49 TXT-FIELD PIC X(25).
```

This structure defines a VARCHAR host variable with the name VARCHAR-FIELD and a length of 25. You cannot use this group item as a host structure; you cannot use the elementary items in the structure as host variables.

For the rules for varying-length string variables, refer to Figure 106 on page 305.

- A group item which defines a host structure is any two-level structure declared in an SQL declare section. The following example is a host structure:

```

01 PROJ-STRCT.
   05 PROJNO          PIC X(6).
   05 ACTNO           PIC S9(4) COMP.
   05 ACSTAFF         PIC S9(9) COMP.
   05 ACSTDATE        PIC X(10).
   05 ACENDATE        PIC X(10).

```

This structure represents the following list of host variables when used in an SQL statement:

```
PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
```

The two following SQL statements are equivalent:

```

EXEC SQL SELECT PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
        INTO :PROJ-STRCT
        FROM PROJ_ACT
        WHERE PROJNO = '100000'

```

```

EXEC SQL SELECT PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
        INTO :PROJNO, :ACTNO, :ACSTAFF, :ACSTDATE, :ACENDATE
        FROM PROJ_ACT
        WHERE PROJNO = '100000'

```

A host structure can be a stand-alone group item or a substructure of a more complex group item. The following example is a complex group item that contains a host structure:

```

01 EMPLOYEE.
   05 EMPNO          PIC X(6).
   05 EMPNAME.
      10 FIRSTNAME   PIC X(12).
      10 MIDINIT     PIC X(1).
      10 LASTNAME    PIC X(15).
   05 WORKDEPT      PIC X(3).
   05 PHONENO       PIC X(4).

```

The group item EMPNAME is a host structure.

You can use the elementary items in the host structure and the elementary items in the group item containing a host structure as host variables. In the previous example, the following elementary items can be used as host variables:

```
EMPNO, FIRSTNAME, MIDINIT, LASTNAME, WORKDEPT, PHONENO
```

You can include a subordinate group item in the host structure to represent a varying-length string element if that group item conforms to the rules for a varying-length string definition. All of the rules previously stated for the definition of varying-length strings also apply in this situation. The following example is a host structure that contains a VARCHAR element:

```

01 EMPNAME.
   05 FIRSTNAME.
      49 FNLEN       PIC S9(4) COMP.
      49 FNTEXT      PIC X(12).
   05 MIDINIT       PIC X(1).
   05 LASTNAME.
      49 LNLEN       PIC S9(4) COMP.
      49 LNTEXT      PIC X(15).

```

The COBOL preprocessor interprets the structure EMPNAME as a host structure containing three elements: FIRSTNAME with data type VARCHAR and length 12, MIDINIT with data type CHAR and length 1, and LASTNAME with data type VARCHAR and length 15.

Note: Any structure that matches the description of a varying-length string definition is interpreted as a varying-length definition and cannot be used as a host structure.

- Third-level host structures are permitted in COBOL to support varying-length strings. The following is an example of varying-length string declarations in host structures:

```

WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMPNAME.
   05 FIRST-NM   PIC X(8).
   05 LAST-NM    PIC X(8).
   05 ADDRESS.
       49 ADD-LEN PIC S9(4) COMP.
       49 ADD-TXT PIC X(200).
EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
EXEC SQL SELECT FIRSTNAME, LASTNAME, ADDRESS
        INTO   :EMPNAME
        FROM   EMPLOYEE
        WHERE  LASTNAME = 'JOHANSON'
END-EXEC

```

In this example, empname is considered by the COBOL preprocessor to be a two-level structure because the structure of address matches that of a VARCHAR data type. As a result, empname may be used in the SELECT statement. If, for example, addlen was changed from "PIC S9(4) COMP" to "PIC S9(9) COMP", the structure of address would no longer match a VARCHAR data type and empname would be considered a three-level structure. As a result, empname could NOT be used in a SELECT statement.

- A host structure field in an SQL statement must be qualified as *structurename.fieldname* instead of *fieldname OF structurename* or *fieldname IN structurename*.

In the declaration below, only DATES and PRODUCT may be used as host structures. ORDERNO and CUSTNUM may be used as scalar host host variables, and may be qualified as CUSTORD.ORDERNO and ORDINFO.CUSTNUM or CUSTORD.ORDINFO.CUSTNUM.

```

WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 CUSTORD.
   03 ORDERNO    PIC X(10).
   03 ORDINFO.
       05 CUSTNUM PIC X(10).
       05 DATES.
           10 ORDDATE PIC X(6).
           10 DELIVDTE PIC X(6).
   03 PRODUCT.
       05 STOCKNO PIC X(10).
       05 QUANTITY PIC X(3).
EXEC SQL END DECLARE SECTION END-EXEC.

```

```

PROCEDURE DIVISION.
  EXEC SQL SELECT STOCKNO, QUANTITY
           INTO :PRODUCT
           FROM ORDER
           WHERE STOCKNO = '1234567890'
  END-EXEC.

```

- The following restrictions apply to level numbers:
 1. Independent elementary items must have a level number of 01 or 77.
 2. Subordinate elementary items must have a level number from 02 to 49.
 3. The outermost group item must have a level number of 01.
 4. Subordinate group items must have a level number from 02 to 49.
 5. Elementary items in a varying-length string definition must have a level number of 49.
 6. Level 66 and level 88 items will be ignored by the preprocessor.
- Except for an indicator array, FILLER is permitted as the name of an elementary item. If FILLER is used as the name of an elementary item, the item will be ignored.
- In addition to the clauses discussed in Figure 106 on page 305, the COBOL preprocessor supports the following clauses in declarations imbedded in the SQL declare section:

```

GLOBAL
EXTERNAL
SYNCHRONIZED
VALUE

```

- Rules for continuation of variable names and COBOL keywords in declaration statements are the same as those described for SQL statements.
- The database manager allows host variable names, statement labels, and SQL descriptor area names of up to 256 characters in length, subject to any COBOL language restrictions mentioned in this appendix.

Note: Due to the restriction on the number of host variables in a statement, host structures with greater than 256 fields will not be allowed,
- You should not give any variable a name beginning with SQL or RDI. These names are reserved for database manager use.
- Comma separators are supported between the clauses of a declaration statement.

Using Host Variables in SQL Statements

When you reference host variables, host structures, structures fields, or indicator arrays in an SQL statement, you must precede each reference by a colon (:). The colon distinguishes these variables from SQL identifiers (such as column names). The colon is not required outside an SQL statement.

Using Long VARCHAR Host Variables

When you code on-line command level application programs in COBOL, be aware of the following CICS/VSE restriction. The length of the working storage plus the length of the TGT (TARGET GLOBAL TABLE) must not exceed 64K bytes.

This restriction only applies when using Long VARCHAR Host Variables because the length of a single long VARCHAR host variable can be up to 32K bytes.

Using Preprocessor Options

Using the QUOTE Parameter

If the COBOL compiler QUOTE option is used or if the QUOTE option has been specified in the CBL statement of COBOL, then the QUOTE (or Q) option of the preprocessor should also be specified. You should use a single quotation mark (') to delineate constants used in embedded SQL statements, regardless of the COBOL compiler QUOTE option.

Using the COB2 Parameter

When the COB2 parameter is specified, certain functions supported by the COBOL II Release 3 compiler, and later, are also supported by the database manager. These functions include:

- Literals can be 160 characters long.
- ENDIF will be generated where appropriate when expanding code for the SQL WHENEVER statement.

In order to make use of these features, you must specify the COB2 option when preprocessing your application. Existing applications that use these features must be reprocessed and recompiled.

Invoking COPYBOOKS

You should not use the COBOL COPY verb to invoke COPYBOOKS that involve SQL host variables. Instead, use the SQL INCLUDE statement to invoke such COPYBOOKS. This arrangement is necessary because the preprocessor is run before the COBOL compiler.

Using the COBRC Parameter

When the COBRC parameter is specified, the preprocessor will generate the statement 'MOVE ZEROS TO RETURN-CODE' after it generates a call to ARIPRDI. This solves the problem of unexpected or invalid return codes being reported after a COBOL II (or VSE IBM COBOL for VSE) application ends. For example, a REXX EXEC may contain several steps which each execute based on a return code from the previous step. If the application programmer has not set the COBOL special register, RETURN-CODE, the return code is not reliable. This new parameter may be used instead of explicitly setting the special register.

Limitations:

1. If the user's COBOL compiler does not support the special register, RETURN-CODE, the application will not compile successfully. COBOL II supports it and new versions of COBOL support it, but old versions do not.
2. If the application sets the special register, RETURN-CODE, and then does an SQL call, the value is not preserved.

```
MOVE 4 TO RETURN-CODE.  
EXEC SQL INSERT INTO MYTABLE VALUES (1,2).  
STOP RUN.
```

The application will end with return code 0 instead of 4 because when the EXEC SQL statement is expanded the last line generated is 'MOVE ZEROS TO RETURN-CODE'.

3. If the user's compiler does not support the special register, RETURN-CODE, but they have declared a variable called RETURN-CODE, the variable will be updated which can cause unexpected results for the application.

Using the TRUNC Compiler Option

For Version 3.2 of COBOL II or later, use the TRUNC(BIN) compiler option, because the system is half-word boundary sensitive. Under this option, receiving fields are truncated only at halfword, fullword, or doubleword boundaries.

Using the INCLUDE Statement

To include the external secondary input, specify the following at the point in the source code where the secondary input is to be included:

```
EXEC SQL INCLUDE text_file_name END-EXEC.
```

The *text_file_name* is a C-Type source member of a VSE library.

The INCLUDE statement can appear anywhere within the File, Linkage, or Working Storage Sections of the Data Division, and anywhere within the Procedure Division, including the Declaratives Section, if one is used. Note that the INCLUDE statement is the only type of SQL statement that is allowed within the Declaratives Section of a Procedure Division.

Using COBOL Variables in SQL: Data Conversion Considerations

COBOL variables used in SQL statements must be type-compatible with the columns of the tables with which they are to be used (stored, retrieved, or compared). Of course, an overflow condition may occur if, for example, an INTEGER data item is retrieved into a PICTURE S9(4) variable, and its current value is too large to fit.

The database manager recognizes the DISPLAY SIGN LEADING SEPARATE (DSLS) attribute for COBOL host variables. It converts input host variables in the DSLS format to the required column format, and output host variables from the column format to DSLS format.

The character data types CHAR, VARCHAR, and LONG VARCHAR are considered compatible. The graphic data types GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC are considered compatible. A varying-length string is automatically converted to a fixed-length string, and a fixed-length string is automatically converted to a varying-length string when necessary. If a varying-length string is converted to a fixed-length string, it is truncated or padded on the right with blanks to the correct length. The system also truncates or pads with blanks if a fixed-length string is assigned to another fixed-length string of a different size (for example, a variable of PICTURE X(12) is stored in a column of type CHAR(18)).

The system also considers the datetime data types to be compatible with character data types (fixed or varying, but not LONG VARCHAR and VARCHAR > 254).

Refer to “Converting Data” on page 48 for a data conversion summary.

Other Coding Considerations

You may want to consider the following points when coding SQL statements and host variable declarations:

- The preprocessor scans past COBOL NOTE-type comments and line comments defined by an asterisk (*) in column 7. It does not recognize line comments identified by a slash (/) in column 7.
- An SQL comment entered in a static SQL statement must be preceded by a blank.
- When performing subtraction in an SQL statement, delimit the minus sign (-) with blanks:

```
      blanks
      | |
      V V
QUANT - :ORDER-AMOUNT
```

- COBOL keywords can be coded in mixed case. For example, Data Division is allowed.

Using DBCS Characters in COBOL

If your program contains DBCS characters, the following sequence of processing is necessary:

- DB2 Server for VSE COBOL Preprocessor
- COBOL Kanji Preprocessor
- CICS/VSE Translator, if necessary
- COBOL Compiler.

The rules for the format and use of DBCS characters in SQL statements are the same for COBOL as for other host languages supported by the system. For a discussion of these rules, see “Using a Double-Byte Character Set (DBCS)” on page 50.

When coding graphic constants in SQL statements, use the SQL format of the graphic constant:

```
G' < XXXX >'
```

Note: N is a synonym for G.

See “Using Graphic Constants” on page 58 for a discussion of graphic constants.

The COBOL preprocessor does not support options for changing the encoding for the < and > characters.

Handling SQL Errors

You can declare the SQLCA return code structure that is required for the system in two ways:

1. You may write:

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

in the Working-Storage Section of your source program. The preprocessor replaces this with a declaration of the SQLCA structure.

2. You may declare the SQLCA yourself in the Working-Storage Section, as shown in Figure 105 on page 303.

```
01 SQLCA.  
 05 SQLCAID      PIC X(8).  
 05 SQLCABC      S9(9) COMPUTATIONAL.  
 05 SQLCODE      PIC S9(9) COMPUTATIONAL.  
 05 SQLERRM.  
   49 SQLERRML   PIC S9(4) COMPUTATIONAL.  
   49 SQLERRMC   PIC X(70).  
 05 SQLERRP      PIC X(8).  
 05 SQLERRD      OCCURS 6 TIMES  
                  PIC S9(9) COMPUTATIONAL.  
  
 05 SQLWARN.  
   10 SQLWARN0   PIC X(1).  
   10 SQLWARN1   PIC X(1).  
   10 SQLWARN2   PIC X(1).  
   10 SQLWARN3   PIC X(1).  
   10 SQLWARN4   PIC X(1).  
   10 SQLWARN5   PIC X(1).  
   10 SQLWARN6   PIC X(1).  
   10 SQLWARN7   PIC X(1).  
   10 SQLWARN8   PIC X(1).  
   10 SQLWARN9   PIC X(1).  
   10 SQLWARNA   PIC X(1).  
 05 SQLSTATE     PIC X(5).
```

Figure 105. SQLCA Structure (in COBOL)

A COBOL program containing SQL statements must have a Working-Storage Section. The meanings of the fields within the SQLCA are discussed in the *DB2 Server for VSE & VM SQL Reference* manual.

In COBOL, the object of a GO TO in the SQL WHENEVER statement must be a section name or an unqualified paragraph name.

You may find that the only variable in the SQLCA you really need is SQLCODE. If this is the case, declare just the SQLCODE variable and invoke NOSQLCA support at preprocessor time.

The number of SQLCODE declarations is not limited by the preprocessor. If a stand-alone SQLCODE is specified, the code inserted by the preprocessor into the COBOL code to expand an EXEC SQL statement will refer to the address of that SQLCODE. The COBOL compiler determines if multiple declarations within a program section are not acceptable. In addition, the COBOL compiler determines which region of the code an SQLCODE declaration refers to.

DB2 Server for VSE does not pass the return code in register 15 on completion of SQL statement processing. The return code and any other information is passed in the SQLCA. Furthermore, if the COBRP preprocessor parameter was not specified, DB2 Server for VSE does not set the return code to zeros on completion of SQL statement processing. If IBM COBOL for VSE or COBOL II is being used, this can cause register 15 to be uninitialized and can contain unpredictable data. This appears as very large return codes when the COBOL application ends. This does not occur with the DOS/VS COBOL compiler. It is the responsibility of the

application programmer to set the return code to something meaningful. The COBOL special register RETURN-CODE should be set before the application program ends.

The simplest method is to code the following lines just before a STOP RUN or a GOBACK statement.

```
MOVE ZERO TO RETURN-CODE.  
STOP RUN.
```

Any return code meaningful to the application can be set. It can also be set to the SQLCODE if desired.

Using Dynamic SQL Statements in COBOL

The COBOL preprocessor lets you use a descriptor area, the SQLDA, to execute dynamically defined SQL statements. (See Chapter 6, "Using Dynamic Statements" on page 159 for more information on dynamic SQL statements and for more information on dynamic SQL statements and the SQLDA.) However, the COBOL preprocessor will not replace the statement EXEC SQL INCLUDE SQLDA with a declaration of the SQLDA structure, as is done with the SQLCA. Instead, EXEC SQL INCLUDE SQLDA would just include the secondary input file SQLDA, as described in "Using the INCLUDE Statement" on page 301.

Before you can use the descriptor area you must properly allocate and initialize it, and you must manage all its address variables. The following example shows how you could define a descriptor area in the COBOL Working-Storage section for five fields:

```
01 DASQL.  
  02 DAID          PIC X(8) VALUE 'SQLDA  '.  
  02 DABC          PIC S9(8) COMP VALUE 13216.  
  02 DAN           PIC S9(4) COMP VALUE 5.  
  02 DAD           PIC S9(4) COMP VALUE 0.  
  02 DAVAR        OCCURS 1 TO 300 TIMES  
                  DEPENDING ON DAN.  
  03 DATYPE       PIC S9(4) COMP.  
  03 DALEN        PIC S9(4) COMP.  
  03 FILLER REDEFINES DALEN.  
    15 SQLPRCSN   PIC X.  
    15 SQLSCALE   PIC X.  
  03 DADATA       POINTER.  
  03 DAIND        POINTER.  
  03 DANAME.  
    49 DANAMEL    PIC S9(4) COMP.  
    49 DANAMEC    PIC X(30).
```

Note: DOS/VS COBOL 3.1 users cannot use the "USAGE IS POINTER" clause implied in this example for the DADATA and DAIND areas. Instead, these areas must be defined with the characteristics of PIC X(4).

The descriptor area must not be declared within the SQL declare section.

The following pseudocode illustrates a use of the descriptor area, adequate for three fields:

- allocate storage for a Descriptor Area of at least size = 3
- set DAN = 3 (number of fields)
- set DAD = 3
- set the rest of the values and pointers in the Descriptor Area
EXEC SQL EXECUTE S1 USING DESCRIPTOR dasql

When decimal data is used, the values of the SQLPRCSN and SQLSCALE field can be determined by declaring additional variables. For example:

```

01 PRCSNN          PIC S9(4) COMP.
01 PRCSNC REDEFINES PRCSNN.
   15 FILLCHAR1    PIC X.
   15 PRCSNCHAR    PIC X.
01 SCALEN          PIC S9(4) COMP.
01 SCALEC REDEFINES SCALEN.
   15 FILLCHAR2    PIC X.
   15 SCALECHAR    PIC X.

```

The following MOVE statements would move the precision and scale of the nth selected item into PRCSNN and SCALEN, respectively:

```

MOVE SQLPRCSN(n) TO PRCSNCHAR.
MOVE SQLSCALE(n) TO SCARECHAR.

```

For COBOL, the string-spec in PREPARE and EXECUTE IMMEDIATE must be in the same format as the SQL VARCHAR data type (you must set the proper length) or a quoted string. If a quoted string is used, its length is limited to 120 characters (the maximum length allowed for COBOL constants). In addition, you cannot use a single (') or double (") quotation mark within a COBOL constant that is the object of a PREPARE or EXECUTE IMMEDIATE statement.

Defining DB2 Server for VSE Data Types for COBOL

Figure 106 (Page 1 of 3). DB2 Server for VSE Data Types for COBOL

Description	DB2 Server for VSE Keyword	Equivalent COBOL Declaration
A binary integer of 31 bits, plus sign.	INTEGER or INT	01 PICTURE S9(9) COMPUTATIONAL.
A binary integer of 15 bits, plus sign.	SMALLINT	01 PICTURE S9(4) COMPUTATIONAL.
A packed decimal number, precision p , scale s ($1 \leq p \leq 31$ and $0 \leq s \leq p$). In storage the number occupies a maximum of 16 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL[(p , s)] or DEC[(p , s)]	01 PICTURE S9(x)[V9(y)] COMPUTATIONAL-3. or or 01 PICTURE S9(x)[V9(y)] DISPLAY SIGN LEADING SEPARATE Where $x + y = p$ and $y = s$

Figure 106 (Page 2 of 3). DB2 Server for VSE Data Types for COBOL

Description	DB2 Server for VSE Keyword	Equivalent COBOL Declaration
A single-precision (4-byte) floating-point number, in short System/390 floating-point format.	REAL or FLOAT(<i>p</i>), 1 ≤ <i>p</i> ≤ 21	COMPUTATIONAL-1.
A double-precision (8-byte) floating-point number, in long System/390 floating-point format.	FLOAT or FLOAT(<i>p</i>), 22 ≤ <i>p</i> ≤ 53 or DOUBLE PRECISION	COMPUTATIONAL-2.
A fixed-length character string of length <i>n</i> where 0 < <i>n</i> ≤ 254.	CHARACTER[(<i>n</i>)] or CHAR[(<i>n</i>)]	01 S PICTURE X(<i>n</i>).
A varying-length character string of maximum length <i>n</i> . If <i>n</i> > 254 or ≤ 32 767, this data type is considered a long field. (See “Using Long Strings” on page 45 for more information.) (Only the actual length is stored in the database.)	VARCHAR(<i>n</i>)	01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(<i>n</i>).
A varying-length character string of maximum length 32 767 bytes.	LONG VARCHAR	01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(<i>n</i>).
A fixed-length string of <i>n</i> DBCS characters where 0 < <i>n</i> ≤ 127.	GRAPHIC[(<i>n</i>)]	01 GNAME PICTURE G(<i>n</i>) [DISPLAY-1].
A varying-length string of <i>n</i> DBCS characters. If <i>n</i> > 127 or ≤ 16383, this data type is considered a long field. (See “Using Long Strings” on page 45 for more information.)	VARGRAPHIC(<i>n</i>)	01 GNAME. 49 GGLEN PICTURE S9(4) COMPUTATIONAL. 49 GGVAL PICTURE G(<i>n</i>) [DISPLAY-1].
A varying-length string of DBCS characters of maximum length 16383.	LONG VARGRAPHIC	01 XNAME. 49 XNAMLEN PICTURE S9(4) COMPUTATIONAL. 49 XNAMVAL PICTURE G(<i>n</i>) [DISPLAY-1].
A fixed or varying-length character string representing a date. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	DATE	01 S PICTURE X(<i>n</i>). or 01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(<i>n</i>).

Figure 106 (Page 3 of 3). DB2 Server for VSE Data Types for COBOL

Description	DB2 Server for VSE Keyword	Equivalent COBOL Declaration
A fixed or varying-length character string representing a time. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIME	01 S PICTURE X(n). or 01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(n).
A fixed or varying-length character string representing a timestamp. The lengths can vary on input and output. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIMESTAMP	01 S PICTURE X(n). or 01 S. 49 S-LENGTH PICTURE S9(4) COMPUTATIONAL. 49 S-VALUE PICTURE X(n).

Notes:

1. USAGE or USAGE IS is optional before COMPUTATIONAL, BINARY, PACKED-DECIMAL, and DISPLAY-1.
2. The word IS can follow PICTURE or PIC.
3. COMPUTATIONAL can be abbreviated COMP. PICTURE can be abbreviated PIC.
4. The following synonyms are supported:
 - COMPUTATIONAL-4 for COMPUTATIONAL
 - BINARY for COMPUTATIONAL
 - PACKED-DECIMAL for COMPUTATIONAL-3
 - N(n) for G(g)
5. INTEGER and SMALLINT data types can have sliding ranges. For example, if you want to declare a SMALLINT variable that you know will remain very small, you could use S9(2) instead of S9(4). Or, you could declare an integer with a range of S9(7) instead of S9(9). However, only the ranges shown in the above table allow for the largest possible values of SMALLINT and INTEGER. Truncation may occur if you declare smaller ranges.
6. For COMPUTATIONAL types, 9's may be repeated rather than using the repetition factors in parentheses (that is, 9999 instead of 9(4)). The same is true for the X's in the character types and the G's in the graphic character types.
7. In DECIMAL data types, precision is the total number of digits. Scale is the number of digits to the right of the decimal point.
8. NUMERIC is a synonym for DECIMAL and, can be used when you are creating or altering tables. In such cases, however, the CREATE or ALTER function will establish the column (or columns) as DECIMAL.
9. When a VALUE clause is used for host variables of the form "PIC S9(4) COMP", the highest value accepted by COBOL is 9999. If you specify the COBOL NOTRUNC option, however, a value up to 32 767 can be *moved* into

the host variable. If host variables are to contain long fields where the length exceeds 9999, the NOTRUNC option must be set.

Using Reentrant COBOL Programs

A reentrant program has the characteristic of dynamic allocation of space for data and save areas. This reentrant characteristic can be used in COBOL programs that use the database manager.

Such programs must follow the COBOL compiler's rules for producing reentrant programs, and must be reprocessed, recompiled, and relinked with the OBJECT file ARIPADR4.

Existing COBOL programs may continue to use ARIPADR until they are recompiled. Thereafter, they must link-edit the OBJECT file ARIPADR4.

Using the DYNAM Compiler Option

The DYNAM option of the IBM COBOL for VSE and VS COBOL II compilers can be used for batch and ICCF applications. "Link-Editing and Loading the Program" on page 132.

Applications using the DYNAM option must have access to the DB2 Server for VSE production library at run time.

CICS/VSE programs do not support the DYNAM option; they must continue to be link-edited with the required extra linkage modules.

Using Stored Procedures

The following example shows how to define the parameters in a stored procedure that uses the GENERAL linkage convention.

```
IDENTIFICATION DIVISION.  
.  
.  
DATA DIVISION.  
.  
.  
LINKAGE SECTION.  
01 PARM1 ...  
01 PARM2 ...  
.  
.  
PROCEDURE DIVISION USING PARM1, PARM2.  
.  
.
```

Figure 107. Stored Procedure - Using GENERAL Linkage Convention

The following example shows how to define the parameters in a stored procedure that uses the GENERAL WITH NULLS linkage convention.

```
IDENTIFICATION DIVISION.  
.  
.  
DATA DIVISION.  
.  
.  
LINKAGE SECTION.  
01 PARM1 ...  
01 PARM2 ...  
01 INDARRAY PIC S9(4) USAGE COMP OCCURS 2 TIMES.  
.  
.  
PROCEDURE DIVISION USING PARM1, PARM2, INDARRAY.  
.  
.
```

Figure 108. Stored Procedure - Using GENERAL WITH NULLS Linkage Convention

Appendix D. Using SQL in FORTRAN

Contents

Using ARIS6FTD, a FORTRAN Sample Program	312
Rules for Using SQL in FORTRAN	312
Placing and Continuing SQL Statements	312
Placing Data Statements	313
Using FORTRAN Common Areas	313
Identifying Rules for Case	313
Declaring Host Variables	313
Embedding SQL Statements	315
Using Host Variables in SQL Statements	315
Using Variable Length Character Strings	315
Using DBCS Characters in FORTRAN	318
Using the INCLUDE Statement	318
Using FORTRAN Variables in SQL: Data Conversion Considerations	318
Handling SQL Errors	318
Handling Program Interrupts	319
Using Dynamic SQL Statements in FORTRAN	319
Restrictions When Using the FORTRAN Preprocessor	320
Defining DB2 Server for VSE Data Types for FORTRAN	321

Using ARIS6FTD, a FORTRAN Sample Program

ARIS6FTD is a FORTRAN language sample program for VSE systems that is shipped with the the DB2 Server for VSE product. It resides on the production disk for the base product. You may find it useful to print this sample program before going through this appendix as the hard copy will provide an illustration for many of the topics discussed here.

Note, for example, how the program satisfies the requirements of the application prolog and epilog. Near the beginning of the program all the host variables are declared, and error handling is defined. Near the logical end of the program, the database changes are rolled back, to assure the database remains consistent for each use of the sample program. For your own applications, of course, you will enter a commit.

The data description statements for the host variables are determined by referring to Figure 111 on page 321. When you are coding your own applications you will need to obtain the data types of the columns that your host variables interact with. This can be done by querying the catalog tables. See the *DB2 Server for VSE & VM SQL Reference* manual for more information on catalog tables.

Rules for Using SQL in FORTRAN

The FORTRAN SQL preprocessor supports programs written for the VS FORTRAN compiler with the LANTLR (77) option specified. Only FIXED-FORM source statements are supported.

If FORTRAN labels are placed on SQL declarative statements, the label will be removed and an information message given.

The FORTRAN preprocessor supports a maximum of 255 program units per input source file (254 subprograms in addition to the main program).

Placing and Continuing SQL Statements

All SQL statements must be placed in columns 7 to 72. Columns 73 to 80 may contain sequence numbers and information; columns 1 to 5 may also contain statement numbers.

The rules for continuation of tokens from one line to the next are the same as the FORTRAN rules for the continuation of words and constants.

An SQL statement may use up to 124 continuation lines in addition to the first line (for a total of 125 lines including blanks and comments). A continuation line can be:

- A continued line (that is, a line that does not have a blank or zero in column 6).
- A blank line
- A comment line.

These lines must fall between the start of the SQL statement and the next statement.

Notes:

1. The maximum length of an SQL statement is 8 192 characters
2. This restriction also applies to FORTRAN IF and ELSE statements
3. A statement is terminated by another statement or by end-of-file.

Placing Data Statements

The FORTRAN Release 3.0 compiler restricts the placement of data statements in FORTRAN programs or subroutines. Some precaution is necessary in order to eliminate the following warning message during compilation of the program or subroutine:

```
WARNING MSGIFX1935I
DATA STATEMENT PRECEDES AN EXPLICIT TYPE STATEMENT
```

During preprocessing, the FORTRAN preprocessor places inline calls at the end of the DB2 Server for VSE declare section, if one exists; otherwise, the calls are placed at the beginning of the program or subroutine. These calls contain data statements that must be preceded by all declares.

If an SQL declare section does not exist, place the following dummy SQL declare section after all other program declares to avoid the above warning message:

```
EXEC SQL BEGIN DECLARE SECTION
EXEC SQL END DECLARE SECTION
```

Since the preprocessor replaces EXEC SQL INCLUDE SQLCA with the declaration of the SQLCA structure, the SQLCA must be included before the declare section.

The FORTRAN preprocessor does not recognize a FUNCTION keyword if it is preceded by a type declaration. The FUNCTION keyword must, therefore, be the first word in the FUNCTION statement.

Using FORTRAN Common Areas

For VSE single user mode, items in a FORTRAN COMMON statement must be initialized in a BLOCK DATA subroutine and the COMMON statement must be assigned a name.

Identifying Rules for Case

Mixed case can be used in your FORTRAN program. The SQL preprocessor will change the lowercase into uppercase, except for text within quotation marks, which will be left in the original case.

Declaring Host Variables

Host variables must be explicitly declared to be used in SQL statements. The following example shows an SQL declare section for a FORTRAN program:

```
EXEC SQL BEGIN DECLARE SECTION (at beginning of section)
      .
      .
(Data description entries for host variables)
      .
      .
EXEC SQL END DECLARE SECTION (at end of section)
```

Place the data description entries for all the host variables within the SQL declare sections. You may use the variables appearing in these SQL declare sections in regular FORTRAN statements as well as in SQL statements.

A host variable declared within the SQL DECLARE SECTION may not be continued. The host variable declaration must appear on a single line in order to be recognized by the preprocessor.

You can also place data description entries for non-host variables in the SQL declare section as the FORTRAN preprocessor ignores data description entries within the SQL declare section that it does not recognize as valid host variable declarations. No error message is generated; instead, the statement is left for the FORTRAN compiler to process. Thus it is possible, though not recommended, to place all data description entries within an SQL declare section.

The rules for declaring variables within SQL declare sections are:

- Host variables must be valid FORTRAN variable names according to the version of the FORTRAN compiler that is being used. FORTRAN host variable names are restricted to a length of 18 bytes.
- Variables named in the SQL declare sections must have data descriptions like those in Figure 111 on page 321.
- Variables cannot be any of the following:
 - Vector or array declarations
 - Constants defined by a PARAMETER statement
 - Any declarations that use expressions to define the length of the variables
 - Character variables declared with an undefined length, such as CHARACTER*(*).
- You should not give any variable a name beginning with SQL, because these names are reserved for database manager use.
- When host variables are declared as INTEGER, and you are using the OPTIMIZE(2) or OPTIMIZE(3) compile option, the host variables should be declared as COMMON.

Under OPTIMIZE 2 or 3, FORTRAN may make register assignments to the program variables if they are not in COMMON storage. Under some circumstances, this can result in the database manager using an inaccurate variable value.

In the following example, NUM must be declared as COMMON if OPTIMIZE 2 or 3 is specified:

```
EXEC SQL DECLARE CURSOR C1 FOR INSERT INTO T1 VALUES (:NUM)
EXEC SQL OPEN C1
DO 20 NUM=1,10
    EXEC SQL PUT C1
20  CONTINUE
*
EXEC SQL CLOSE C1
```

- Only the NONE value of the AUTODBL FORTRAN compile option is supported. AUTODBL changes the precision of declared variables without altering the source code. The preprocessor runs before the FORTRAN compiler and interprets variable types based strictly on their declaration.

A host variable must be declared earlier than the first use of the variable in an SQL statement in the program.

Embedding SQL Statements

You must precede each SQL statement in your program with EXEC SQL. No delimiter should be used at the end of each statement.

FORTRAN source statements and SQL statements cannot be contained on the same line or within the same continued statement, except when an SQL statement is used as the imperative statement of a logical IF. Also, only one SQL statement can be contained in a single line, or within the same continued statement.

Using Host Variables in SQL Statements

When you place host variables within an SQL statement, you must precede each one by a colon (:), to distinguish it from the SQL identifiers (such as a column name). When you place a host variable outside of an SQL statement, do not use a colon.

A host variable can represent a data value, but not an SQL identifier. For example, you cannot assign a character constant such as 'MUSICIANS' to a host variable, and then use that host variable in a CREATE TABLE statement to represent the table name. This pseudocode sequence is invalid:

```
IT = 'MUSICIANS'  
CREATE TABLE :IT (NAME ...
```

Incorrect

Using Variable Length Character Strings

FORTRAN does not support variable length character strings (VARCHAR, LONG VARCHAR). However, it is possible to circumvent this restriction in the following way:

1. Declare INTEGER*2 to contain the length of the string
2. Declare a CHARACTER*(length) string of data
3. Declare a CHARACTER*(2 + length of string)
4. Declare a COMMON block containing (1) and (2) above
5. Use the EQUIVALENCE statement (name of (1) above, name of (3) above)
6. Specify a DATA BLOCK subroutine to initialize (1) and (2) above
7. When referencing the string in an SQL statement, use (3) above.
8. After preprocessing the FORTRAN program (but before compilation), change all occurrences of the data code for the variable(s) in the input or output data structure(s) from the CHARACTER data code to the corresponding VARCHAR or LONG VARCHAR data code. For information on how to interpret the data codes returned in SQLTYPE, see the *DB2 Server for VSE & VM SQL Reference* manual.

Figure 109 on page 317 shows an example of how to INSERT a row into the INVENTORY table using a VARCHAR variable for description.

Note: It is necessary to set the length field (STRNGL) to the corresponding length of the character string (STRING) before the insert statement is executed.

When the character string is fetched, the first two bytes of the string (STRNGW) contain the length. The variable STRNGL determines the length.

```

C*** DB2 Server for VSE STATEMENT ***
C   EXEC SQL BEGIN DECLARE SECTION
      CHARACTER ID*8
      CHARACTER PW*8
      INTEGER*2      STRNGL
      CHARACTER*24   STRING
      CHARACTER*26   STRNGW
      COMMON /SDATA/ STRNGL,STRING
      EQUIVALENCE    (STRNGL,STRNGW)
      ...
      ...
C*** DB2 Server for VSE STATEMENT ***
C   EXEC SQL END DECLARE SECTION
      ...
      ...
C*** DB2 Server for VSE STATEMENT ***
C   EXEC SQL INSERT INTO SQLDBA.ACTIVITY
C   1          VALUES(190, 'TSTSYS',:STRNGW)
C
      SQI002( 3, 1) = 1
      SQI002( 1, 2) = 452 * SQSHHW + 26 ---> Change 452 to 448
      SQI002( 2, 2) = SQLADD(STRNGW)
      SQI002( 3, 2) = 0
      SQCALL = 'EXECUTE '
      SQSTMT = -1
      SQLTYP = '0'

      SQLCTL(1) = SQLADD ( SQCALL )
      SQLCTL(2) = SQLADD ( SQCOLL )
      SQLCTL(3) = SQLADD ( SQPROG )
      SQLCTL(4) = SQLADD ( SQSTMT )
      SQLCTL(5) = SQLADD ( SQI002 )
      SQLCTL(6) = 0
      SQLCTL(7) = 0
      SQLCTL(8) = 8
      SQLCTL(9) = SQLADD ( SQLISL )
      SQLCTL(10) = SQLADD ( SQLDAT )
      SQLCTL(11) = SQLADD ( SQLTIM )
      SQLCTL(12) = SQLADD ( SQLCNT )
      SQLCTL(13) = SQLADD ( SQLTYP )
      CALL ARIFOR ( SQLCTL )
      ...
      ...
      END

*****
*                               BLOCK DATA SUBROUTINE
*****
      BLOCK DATA
      COMMON /SDATA/ STRGNL,STRING
      INTEGER*2 STRGNL/3/
      CHARACTER*24 STRING/'SYSTEM TESTING'/
      END

```

Figure 109. Using a VARCHAR Variable

Using DBCS Characters in FORTRAN

The rules for the format and use of DBCS characters in SQL statements are the same for FORTRAN as for other host languages supported by the system. For a discussion of these rules, see “Using a Double-Byte Character Set (DBCS)” on page 50.

FORTRAN does not provide a way to define graphic host variables. If you want to add graphic data to or retrieve it from DB2 Server for VSE tables, you must execute the affected statements dynamically. By doing so, the data areas that are referenced by each statement can be described in an SQLDA. In the SQLDA, you must set the data type of the areas containing graphic data to one of the graphic data types. (For a discussion of the SQLDA, refer to the *DB2 Server for VSE & VM SQL Reference* manual.)

Using the INCLUDE Statement

To include the external secondary input, specify the following at the point in the source code where the secondary input is to be included:

```
EXEC SQL INCLUDE text_name
```

Text_name is the G-Type source member of a VSE library. is the file name of a CMS file (with a “FORTCOPY”

Using FORTRAN Variables in SQL: Data Conversion Considerations

Host variables must be type-compatible with the columns with which they are to be used.

A column of type INTEGER, SMALLINT, or DECIMAL is compatible with a FORTRAN variable of INTEGER, INTEGER*2, or INTEGER*4. Of course, an overflow condition may occur if, for example, an INTEGER data item is retrieved into an INTEGER*2 variable, and its current value is too large to fit.

Fixed-length and varying-length character data (CHAR, VARCHAR, and LONG VARCHAR) are considered compatible. A varying-length string is automatically converted to a fixed-length string, and a fixed-length string is automatically converted to a varying-length string, when necessary. If a varying-length string is converted to a fixed-length string, it is truncated or padded on the right with blanks to the correct length.

The database manager also considers the datetime data types to be compatible with character data types (CHAR and VARCHAR, but not LONG VARCHAR and VARCHAR > 254).

Refer to “Converting Data” on page 48 for a data conversion summary.

Handling SQL Errors

There are two ways to declare the return code structure (called SQLCA):

1. You may write:

```
EXEC SQL INCLUDE SQLCA
```

in your source program. The preprocessor replaces this with the declaration of the SQLCA structure.

2. You may declare the SQLCA structure directly, as shown in Figure 110 on page 319.

```
INTEGER*4      SQLCOD,  
*             SQLERR(6),  
*             SQLTXL*2  
COMMON /SQLCA1/ SQLCOD,SQLERR,SQLTXL  
  
CHARACTER     SQLERP*8,  
*            SQLWRN(0:10),  
*            SQLTXT*70,  
*            SQLSTT*5  
COMMON /SQLCA2/ SQLERP,SQLWRN,SQLTXT,SQLSTT
```

Figure 110. SQLCA Structure (in FORTRAN)

The SQLCA must not be declared within the SQL declare section. The meanings of the fields within the SQLCA are discussed in the *DB2 Server for VSE & VM SQL Reference* manual.

You may find that the only variable in the SQLCA you really need is SQLCODE. If this is the case, declare just the SQLCOD variable and invoke NOSQLCA support at preprocessor time.

Note: FORTRAN requires SQLCOD instead of SQLCODE.

The number of SQLCOD declarations is not limited by the preprocessor. If a stand-alone SQLCOD is specified, the code inserted by the preprocessor into the FORTRAN code to expand an EXEC SQL statement will refer to the address of that SQLCOD. The FORTRAN compiler determines if multiple declarations within a program section are not acceptable. In addition, the FORTRAN compiler determines which region of the code an SQLCOD declaration refers to.

Handling Program Interrupts

If a program interrupt occurs and the database manager is unaware of it, you may get unexpected results. To allow the system to process the interrupt, specify the run time options NOSTAE and NOSPIE. These options are only available in Version 2 of FORTRAN.

Using Dynamic SQL Statements in FORTRAN

The FORTRAN preprocessor lets you use a descriptor area, the SQLDA, to execute dynamically defined SQL statements. (See Chapter 6, “Using Dynamic Statements” on page 159 for information on dynamic SQL statements and the SQLDA.) However, the FORTRAN preprocessor will not replace the statement EXEC SQL INCLUDE SQLDA with a declaration of the SQLDA structure, as is done with the SQLCA. Instead EXEC SQL INCLUDE SQLDA would just include the secondary input file SQLDA, as described in the section “Using the INCLUDE Statement” on page 318.

Before you can use the descriptor area you must properly allocate and initialize it, and you must manage all its address variables. The following example shows how you could define the descriptor area in FORTRAN for three fields:

```

CHARACTER*8 DAID
INTEGER*4 DABC
INTEGER*2 DASQLN,
* DAD,
* DATYPE_1, DATYPE_2, DATYPE_3,
* DALEN_1, DALEN_2, DALEN_3,
* DANLN_1, DANLN_2, DANLN_3
INTEGER*4 DADATA_1, DADATA_2, DADATA_3,
* DAIND_1, DAIND_2, DAIND_3
CHARACTER*30 DANAME_1, DANAME_2, DANAME_3

COMMON /DASQL/ DAID, DABC, DAN, DAD,
* DATYPE_1, DALEN_1, DADATA_1, DAIND_1, DANLN_1, DANAME_1,
* DATYPE_2, DALEN_2, DADATA_2, DAIND_2, DANLN_2, DANAME_2,
* DATYPE_3, DALEN_3, DADATA_3, DAIND_3, DANLN_3, DANAME_3

```

The descriptor area must not be declared within the SQL declare section.

The following pseudocode illustrates the use of a descriptor area, adequate for three fields:

```

- allocate storage for a Descriptor Area of at least size = 3
- set DAN = 3 (number of fields)
- set DAD = 3
- set the rest of the values and pointers in the Descriptor Area
EXEC SQL EXECUTE S1 USING DESCRIPTOR dasql

```

Restrictions When Using the FORTRAN Preprocessor

The FORTRAN preprocessor is an extended dynamic preprocessor that uses the NOMODIFY and DESCRIBE options of the extended CREATE PACKAGE statement. The other extended CREATE PACKAGE options that are used are taken from the parameters specified when invoking the preprocessor.

FORTRAN programs are preprocessed and executed using extended dynamic SQL. Those that are preprocessed with the DB2 Server for VSE FORTRAN preprocessor must, therefore, comply with the same restrictions that apply to extended dynamic SQL, or programs preprocessed or executed using extended dynamic SQL.

The following is a partial list of restrictions when using the FORTRAN preprocessor.

- The BIND preprocessing parameter is ignored by the FORTRAN preprocessor.
- When declaring a dynamic cursor, if you are using the following format of the PREPARE statement, you must code it in your program before the DECLARE CURSOR statement:

```
PREPARE statement_name FROM string_constant
```

This restriction does not apply when using the following format of the PREPARE statement:

```
PREPARE statement_name FROM host_variable
```

Refer to “Mapping Extended Dynamic Statements to Static and Dynamic Statements” on page 195 for details about mapping extended dynamic statements to non-extended dynamic statements. for a

Defining DB2 Server for VSE Data Types for FORTRAN

Figure 111 (Page 1 of 2). DB2 Server for VSE Data Types for FORTRAN

Description	DB2 Server for VSE Keyword	Equivalent FORTRAN Declaration
A binary integer of 31 bits, plus sign.	INTEGER or INT	INTEGER INTEGER*4
A binary integer of 15 bits, plus sign.	SMALLINT	INTEGER*2
A packed decimal number, precision p , scale s ($1 \leq p \leq 31$ and $0 \leq s \leq p$). In storage the number occupies a maximum of 16 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL[(p , s)] or DEC[(p , s)](1) 1	Not supported.
A single-precision (4- byte) floating-point number, in short System/390 floating-point format.	REAL or FLOAT(p), $1 \leq p \leq 21$	REAL REAL*4
A double-precision (8- byte) floating-point number, in long System/390 floating-point format.	FLOAT or FLOAT(p), $22 \leq p \leq 53$ or DOUBLE PRECISION	REAL*8 DOUBLE PRECISION DOUBLEPRECISION
A fixed-length character string of length n where $0 < n \leq 254$.	CHARACTER[(n)] or CHAR[(n)]	CHARACTER CHARACTER* n
A varying-length character string of maximum length n . If $n > 254$ but ≤ 32767 , this data type is considered a long field. (See “Using Long Strings” on page 45 for more information.)	VARCHAR(n)	Not supported.
A varying-length character string of maximum length 32765 bytes (two bytes less than the DB2 Server for VSE maximum, because of the length field). (Character strings ≥ 255 are not supported in FORTRAN releases prior to Release 1.3.)	LONG VARCHAR	Not supported.
A fixed-length string of n DBCS characters where $0 < n \leq 127$.	GRAPHIC[(n)]	Not supported.
A varying-length string of n DBCS characters. If $n > 127$ but ≤ 16383 , this data type is considered a long field. (See “Using Long Strings” on page 45 for more information.)	VARGRAPHIC(n)	Not supported.

Figure 111 (Page 2 of 2). DB2 Server for VSE Data Types for FORTRAN

Description	DB2 Server for VSE Keyword	Equivalent FORTRAN Declaration
A varying-length string of DBCS characters of maximum length 16383.	LONG VARGRAPHIC	Not supported.
A fixed-length character string representing a date. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	DATE	CHARACTER CHARACTER*n No varying-length equivalent is supported.
A fixed-length character string representing a time. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIME	CHARACTER CHARACTER*n No varying-length equivalent is supported.
A fixed-length character string representing a timestamp. The lengths can vary on input and output. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIMESTAMP	CHARACTER CHARACTER*n No varying-length equivalent is supported.

Notes:

1. NUMERIC is a synonym for DECIMAL and can be used when creating or altering tables. In such cases, however, the CREATE or ALTER function establishes the column (or columns) as DECIMAL.

An * length specification can also be used to override a length specification associated with the initial keyword. The following are examples:

Specification	Valid	Invalid (ignored)
INTEGER VAR001,VAR002(2)	VAR001 4 bytes	VAR002
INTEGER*2 VAR001*4,VAR002	VAR001 4 bytes VAR002 2 bytes	
INTEGER*4 VAR001*2/10,VAR002*4	VAR001 2 bytes VAR002 4 bytes	
INTEGER*5 VAR001*2,VAR002*4		VAR001,VAR002
REAL VAR001*8,VAR002	VAR001 8 bytes VAR002 4 bytes	
REAL*8 VAR001,VAR002*4,VAR003	VAR001 8 bytes VAR002 4 bytes VAR003 8 bytes	
DOUBLE PRECISION VAR001,VAR002*4	VAR001 8 bytes VAR002 4 bytes	
REAL*8 VAR001(10,10)*4,VAR002	VAR002 8 bytes	VAR001
REAL*16 VAR001,VAR002*4,VAR003*8	VAR002 4 bytes VAR003 8 bytes	VAR001

Specification	Valid	Invalid (ignored)
CHARACTER VAR1,VAR2*80	VAR1 1 byte VAR2 80 bytes	
CHARACTER*10 VAR1,VAR2*80	VAR1 10 bytes VAR2 80 bytes	
CHARACTER*500 VAR1(5),VAR2*1	VAR2 1 byte	VAR1

Appendix E. Using SQL in PL/I

Contents

Using ARIS6PLD, a PL/I Sample Program	326
Rules for Using SQL in PL/I	326
Placing and Continuing SQL Statements	326
Delimiting SQL Statements	326
Using the INCLUDE Statement	327
Declaring Static External Variables	327
Identifying Rules for Case	327
Declaring Host Variables	327
Using Host Variables in SQL Statements	330
Using PL/I Variables in SQL: Data Conversion Considerations	330
Using DBCS Characters in PL/I	331
Using SQL Statements in PL/I Subroutines	331
Coding the SIZE Parameter in VSE JCL	332
Handling SQL Errors	332
Handling Program Interrupts	333
Using Dynamic SQL Statements in PL/I	334
Defining DB2 Server for VSE Data Types for PL/I	336
Using Stored Procedures	337

Using ARIS6PLD, a PL/I Sample Program

ARIS6PLD is a PL/I language sample program for VSE systems that is shipped with the DB2 Server for VSE product. It resides on the production disk for the base product. You may find it useful to print this sample program before going through this appendix as the hard copy will provide an illustration for many of the topics discussed here.

You can learn most of the rules for using SQL within PL/I just by scanning through the program. Note, in particular, how the program satisfies the requirements of the application prolog and epilog. Near the beginning of the program all the host variables are declared and error handling is defined. Near the logical end of the program, the database changes are rolled back, to assure the database remains consistent for each use of the sample program. For your own applications, of course, you will enter a commit.

The DCL statements for the host variables are determined by referring to Figure 116 on page 336. That figure gives the PL/I representation for each of the DB2 Server for VSE data types. When you are coding your own applications, you will need to obtain the data types of the columns that your host variables interact with. This can be done by querying the catalog tables, which are described in the *DB2 Server for VSE & VM SQL Reference* manual.

Rules for Using SQL in PL/I

Placing and Continuing SQL Statements

All statements in your PL/I program, including SQL statements, must be contained in columns 2 through 72 of your source deck. Normal PL/I continuation rules apply.

Continuation of tokens (the basic syntactical units of a language) is allowed from one line to the next, by coding the first part of the token up to column 72 on the line to be continued and coding the next part of the token from column 2 on the continuation line. If either column 72 of the continued line or column 2 of the continuation line is blank, the token is not continued.

See the *DB2 Server for VSE & VM SQL Reference* manual for a discussion on tokens.

Delimiting SQL Statements

Delimiters are required on all SQL statements to help the database manager distinguish them from regular PL/I statements. You must precede each SQL statement in your program with EXEC SQL, and end each statement with a semicolon. EXEC and SQL must be on the same line, with only blanks separating them (no in-line host language or SQL comments).

Within SQL statements, host language and SQL comments are allowed anywhere that blanks are allowed. However, there should not be any host language or SQL comments within SQL statements that are dynamically defined and executed.

An SQL statement cannot be followed on the same line by another SQL statement, a normal PL/I statement, or a host language comment. When you preprocess a program containing such a combination, the trailing statements or host language comments will not appear in the SYSLST listing.

Using the INCLUDE Statement

To include external secondary input, specify the following at the point in the source code where the secondary input is to be included:

```
EXEC SQL INCLUDE text_file-name;
```

The *text_file*-name is the member name of a P-Type source member of a VSE library.

Declaring Static External Variables

A declaration for a variable with the attributes STATIC and EXTERNAL must also have the attribute INITIAL. If it does not, the declaration generates a common CSECT, which the database manager cannot handle.

PL/I programming using “DEFAULT RANGE (*) STATIC” gives an error message. The preprocessor builds control blocks that are incompatible with this statement.

Identifying Rules for Case

The keywords “EXEC SQL” must appear in uppercase in your PL/I program. The rest of an SQL statement can be in mixed case, but will be interpreted as uppercase, except for text within quotation marks, which will be left in the original case.

Declaring Host Variables

You must declare all host variables in an SQL declare section. For a description of an SQL declare section, refer to “Declaring Variables That Interact with the Database Manager” on page 6.

Declare host variables in the source file before the first use of the variable in an SQL statement. You can use the following types of variables in an SQL statement:

- Scalar variables
- Structure variables
- Structure elements
- Array variables

For information on the use of these variables in an SQL statement, refer to “Using Host Variables” on page 54 and “Using Host Structures” on page 55.

Note: You can declare non-host variables in an SQL declare section; however, declarations that do not conform to DB2 Server for VSE declaration rules may return errors.

The declaration of a host variable is subject to the following rules:

- You can use scalar variables and structure elements as main variables. You can also use them as indicator variables if they are declared with a data type of short integer.

- The only arrays accepted by the PL/I preprocessor are arrays of short integer elements. These arrays may be used as indicator arrays only. The following example is an indicator array:

```
DCL IND_ARRAY(10) BINARY FIXED(15);
```

Indicator array elements cannot be used as main or indicator variables.

- A structure variable (which defines a host structure) is any two-level structure declared in an SQL declare section. The following example is a host structure:

```
DCL 01 PROJ_STRCT,
    05 PROJNO          CHAR(6),
    05 ACTNO           BINARY FIXED(15),
    05 ACSTAFF         BINARY FIXED(31),
    05 ACSTDATE        CHAR(10),
    05 ACENDATE        CHAR(10);
```

This structure represents the following list of host variables when used in an SQL statement:

```
PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
```

In other words, the two following SQL statements are equivalent:

```
EXEC SQL SELECT PROJNO, ACTNO, ACTSTAFF, ACSTDATE, ACENDATE
        INTO :PROJ_STRCT
        FROM PROJ_ACT
        WHERE PROJNO = '100000'
```

```
EXEC SQL SELECT PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
        INTO :PROJNO, :ACTNO, :ACSTAFF, :ACSTDATE, :ACENDATE
        FROM PROJ_ACT
        WHERE PROJNO = '100000'
```

A host structure can either be a stand-alone structure or a substructure of a more complex structure. The following example is a complex structure that contains a host structure:

```
DCL 01 EMPLOYEE,
    05 EMPNO          CHAR(6),
    05 EMPNAME
        10 FIRSTNAME  CHAR(12),
        10 MIDINIT    CHAR(1),
        10 LASTNAME   CHAR(15),
    05 WORKDEPT      CHAR(3),
    05 PHONENO       CHAR(4);
```

The structure EMPNAME is a host structure.

You can use the elements of the host structure and the elements of a complex structure containing a host structure as host variables. In the previous example, EMPNO, FIRSTNAME, MIDINIT, LASTNAME, WORKDEPT, and PHONENO can all be used as host variables.

- DCL or DECLARE must be the first character sequence on the line, but cannot start in column 1. You can, however, have a carriage control character in column 1. Otherwise, the line is ignored. (You can place inline host language comments anywhere after the DECLARE or DCL keyword, and you can continue these comments over multiple lines.)
- DECLARE statements can be continued on additional lines, but you cannot have more than one DECLARE statement on the same line. All DECLARE

statements must end with a semicolon. Rules for continuation of variable names and PL/I keywords are the same as those described for SQL statements.

- Declare only one host variable per DCL or DECLARE statement. If you declare multiple variables, only the first variable is recognized; the others are ignored. For example:

```
DCL AA FIXED BIN(15) INIT(7),  
    BB CHAR(7),  
    CC BINARY FLOAT(53);
```

BB and CC are ignored.

The next rule provides one exception to this limitation.

- Factoring of scalar variable names, structure element names and indicator array names is supported. For example, the following declarations are valid:

```
DCL (X,Y,Z) BINARY FIXED(31);
```

```
DCL (ARR1(10), ARR2(5), ARR3(6)) BINARY FIXED (15);
```

```
DCL 01 STUCT,  
    05 (FLD1, FLD2, FLD3) CHAR(10),  
    05 FLD4 CHAR(5);
```

- In addition to the attributes discussed in Figure 116 on page 336, the PL/I preprocessor also supports the following attributes in declarations imbedded in an SQL declare section:

```
ALIGNED  
UNALIGNED  
INTERNAL  
EXTERNAL  
STATIC  
AUTOMATIC  
DEFINED  
CONTROLLED  
CONNECTED  
INITIAL
```

- In PL/I, the BASED and LIKE functions are not permitted in host structure declarations.
- You cannot duplicate variable names in a single source file even if they are in different blocks or functions. The PL/I preprocessor defines a duplicate as any name that cannot be referenced unambiguously when fully qualified.
- You should not declare variables whose names begin with SQL or RDI, because these names are reserved for database manager use.
- The database manager allows host variable names, statement labels, and SQL descriptor area names of up to 256 characters in length, subject to any PL/I language restriction mentioned in this appendix.

You can have a label on the “EXEC SQL BEGIN DECLARE SECTION;”, but not on the “EXEC SQL END DECLARE SECTION;”. If you do place a label on this

statement, the preprocessor does not recognize it and assumes that the SQL declare section has not ended.

When placing host language comments after either of these statements, make sure the comment ends on the same line. If it does not, PL/I compiler errors result.

Note: Other program variables can also be declared as usual outside the SQL declare section. The previous restrictions do not apply to non-SQL declarations.

In the declaration below, only DATES and PRODUCTS may be used as host structures. ORDERNO and CUSTNUM may be used as scalar host variables and may be qualified as CUSTORD.ORDERNO and ORDINFO.CUSTNUM or CUSTORD.ORDINFO.CUSTNUM.

```
EXEC SQL BEGIN DECLARE SECTION;
  DCL 1 CUSTORD,
      2 ORDERNO CHAR(10),
      2 ORDINFO,
      3 CUSTNUM CHAR(10),
      3 DATES,
      5 ORDDATE CHAR(6),
      5 DELIVDTE CHAR(6),
      2 PRODUCT,
      3 STOCKNO CHAR(10),
      3 QUANTITY CHAR(3);
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT STOCKNO, QUANTITY
  INTO :PRODUCT
  FROM ORDER
  WHERE STOCKNO = '1234567890';
```

Using Host Variables in SQL Statements

When you reference host variables, host structures, structure fields or indicator arrays in an SQL statement, you must precede each reference by a colon (:). The colon distinguishes these variables from SQL identifiers (such as column names). The colon is not required outside an SQL statement.

Using PL/I Variables in SQL: Data Conversion Considerations

Host variables must be type-compatible with the columns with which they are to be used. For example, if you want to compare a program variable with the QONHAND column of the database, and the data type of QONHAND is INTEGER, you should declare the program variable BIN FIXED(31), BIN FIXED(15), BIN FLOAT, FLOAT BIN, or FIXED DECIMAL(10). (Refer to "Assigning Data Types When the Column Is Created" on page 43 for details on the FLOAT data type.)

The database manager considers the numeric data types compatible, as well as the character string data types (CHAR, VARCHAR, and LONG VARCHAR, including strings of different declared lengths), and the graphic string data types (GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC). Of course, an overflow condition may result if, for example, you assign a 31-bit integer to a 15-bit integer and the current value of the 31-bit integer is too large to fit in 15 bits. Truncation also occurs when a decimal number having a scale greater than zero is assigned to an integer. In general, overflow occurs when significant digits are lost, and truncation occurs when nonsignificant digits are lost.

The datetime data types are also considered compatible with character data types (CHAR, and VARCHAR, but not LONG VARCHAR and VARCHAR > 254).

Refer to “Converting Data” on page 48 for a data conversion summary.

Using DBCS Characters in PL/I

The rules for the format and use of DBCS characters in SQL statements are the same for PL/I as for other host languages supported by the system. For a discussion of these rules, see “Using a Double-Byte Character Set (DBCS)” on page 50.

When using the string-constant format of the PREPARE or EXECUTE IMMEDIATE statement, if the statement in the string-constant contains DBCS characters, you must append an M to the string-constant. For example:

```
EXEC SQL PREPARE S13 FROM
'SELECT TRANSLATE(''1aabb'') || ''1< AB >'' FROM SYSTEM.SYSCCSIDS'M;
```

When coding graphic constants in static SQL statements, use one of the following PL/I formats of the graphic constant:

1. '< XXXX >'G
2. '<@' XXXX '@'@G>

Note: N is a synonym for G.

When coding graphic constants in dynamically executed SQL statements, use the SQL format of the graphic constant (that is, G'< XXXX >'). Refer to “Using Graphic Constants” on page 58 for a discussion of graphic constants.

Using SQL Statements in PL/I Subroutines

The first SQL statement encountered in a sequential scan of your program by the PL/I preprocessor that requires an in-line call to the resource adapter results in the generation of control blocks SQLTIE and RDIEXT, and other declarations commonly used by internal DB2 Server for VSE code that is associated with the remaining SQL statements in your program. If your program structure involves SQL statements in multiple procedures, you must maintain structures so that the SQLTIE and RDIEXT are addressable by all other SQL statement occurrences in your program.

Figure 112 on page 332 represents an incorrect structure.

```

A: PROC OPTIONS(MAIN);
    .
    .
    CALL B;
    CALL C;
    .
    .
B: PROC;
    .
    .
    EXEC SQL CONNECT.....
    EXEC SQL DECLARE C1 CURSOR....
    EXEC SQL OPEN C1 ...
    .
    .
    END B;
C: PROC;
    .
    .
    EXEC SQL DECLARE C2 CURSOR....
    EXEC SQL OPEN C2 .....
    .
    .
    END C;
    .
    .
    END A;

```

Figure 112. Incorrect PL/I Program Structure

SQLTIE and RDIEXT will be generated from the CONNECT in B, but it is not addressable from C, where other SQL statements appear. This can be solved by putting the CONNECT statement in A, where it will cause SQLTIE and RDIEXT to be generated at a place that is addressable by both B and C.

Coding the SIZE Parameter in VSE JCL

When executing PL/I application programs in VSE single user mode, specify SIZE=750K, not SIZE=AUTO, in the EXEC job control statement.

Handling SQL Errors

There are two ways to declare the return code structure (called SQLCA):

1. You can write the following statement in your source program:

```
EXEC SQL INCLUDE SQLCA;
```

The preprocessor replaces this with the declaration of the SQLCA structure.

2. You may declare the SQLCA structure directly, as shown in Figure 113 on page 333.

```

DCL 1 SQLCA,
    2 SQLCAID CHAR(8),
    2 SQLCABC BIN FIXED(31),
    2 SQLCODE BIN FIXED(31),
    2 SQLERRM CHAR(70) VAR,
    2 SQLERRP CHAR(8),
    2 SQLERRD (6) BIN FIXED(31),
    2 SQLWARN,
        3 SQLWARN0 CHAR(1),
        3 SQLWARN1 CHAR(1),
        3 SQLWARN2 CHAR(1),
        3 SQLWARN3 CHAR(1),
        3 SQLWARN4 CHAR(1),
        3 SQLWARN5 CHAR(1),
        3 SQLWARN6 CHAR(1),
        3 SQLWARN7 CHAR(1),
        3 SQLWARN8 CHAR(1),
        3 SQLWARN9 CHAR(1),
        3 SQLWARNA CHAR(1),
    2 SQLSTATE CHAR(5);

```

Figure 113. SQLCA Structure (in PL/I)

The SQLCA must not be declared within the SQL declare section. The meanings of the fields in the SQLCA are discussed in the *DB2 Server for VSE & VM SQL Reference* manual.

You may find that the only variable in the SQLCA you really need is SQLCODE. If this is the case, declare just the SQLCODE variable, and invoke NOSQLCA support at preprocessor time.

The number of SQLCODE declarations is not limited by the preprocessor. If a stand-alone SQLCODE is specified, the code inserted by the preprocessor into the PL/I code to expand an EXEC SQL statement will refer to the address of that SQLCODE. The PL/I compiler determines if multiple declarations within a program section are not acceptable. In addition, the PL/I compiler determines which region of the code an SQLCODE declaration refers to.

Handling Program Interrupts

If a program interrupt occurs and the database manager is unaware of it, you may receive unexpected results. To allow the system to process the interrupt, include the following declaration statement **after** the “EXEC SQL END DECLARATION SECTION” statement:

```
DCL PLIXOPT CHAR(20) VAR INIT('TRAP(OFF)') STATIC EXTERNAL;
```

If your PL/I compiler is NOT Language Environment enabled, add the following statement instead:

```
DCL PLIXOPT CHAR(20) VAR INIT('NOSTAE,NOSPIE') STATIC EXTERNAL;
```

Using Dynamic SQL Statements in PL/I

You may need to declare an SQLDA structure to execute dynamically defined SQL statements. You can have the system include the structure automatically by specifying:

```
EXEC SQL INCLUDE SQLDA;
```

in your source code, or by directly coding the structure as shown in Figure 114.

```
DCL 1 SQLDA BASED(SQLDAPTR),
    2 SQLDAID CHAR(8),
    2 SQLDABC BIN FIXED(31),
    2 SQLN BIN FIXED(15),
    2 SQLD BIN FIXED(15),
    2 SQLVAR(SQLSIZE REFER(SQLN)),
        3 SQLTYPE BIN FIXED(15),
        3 SQLLEN BIN FIXED(15),
        3 SQLDATA PTR,
        3 SQLIND PTR,
        3 SQLNAME CHAR(30) VAR;
DCL SQLSIZE BIN FIXED(15);
DCL SQLDAPTR PTR;
```

Figure 114. SQLDA Structure (in PL/I)

The SQLDA must not be declared within the SQL declare section. See the *DB2 Server for VSE & VM SQL Reference* manual for more information on the individual fields within the SQLDA.

In addition to the structure above, you should declare an additional mapping for the same area. The SQLPRCSN and SQLSCALE fields of the second mapping are used when decimal data is used. Figure 115 shows this mapping.

Figure 115 (Page 1 of 2). SQLDAX Structure (in PL/I)

```
DCL 1 SQLDAX BASED(SQLDAPTR),
    2 SQLDAIDX CHAR(8),
    2 SQLDABCX BIN FIXED(31),
    2 SQLNX BIN FIXED(15),
    2 SQLDX BIN FIXED(15),
    2 SQLVARX(SQLSIZE REFER(SQLNX)),
        3 SQLTYPEX BIN FIXED(15),
        3 SQLPRCSN format 1 or format 2,
        3 SQLSCALE format 1 or format 2,
        3 SQLDATA PTR,
        3 SQLINDX PTR,
        3 SQLNAMEX CHAR(30) VAR;
```

The SQLPRCSN and SQLSCALE fields can be declared in one of two formats.

```
Format 1:    3 SQLPRCSN BIT(8),
              3 SQLSCALE BIT(8),
```

The fields must be set by bit 8 strings. For example, for a precision of 5 and scale of 2, the following assignments are required:

```
SQLDAPTR->SQLPRCSN = '00000101'B
SQLDAPTR->SQLSCALE = '00000010'B
```


Figure 115 (Page 2 of 2). SQLDAX Structure (in PL/I)

```
Format 2:      3 SQLPRCSN CHAR(1),
              3 SQLSCALE CHAR(1),
```

This format requires the declaration of additional variables. These are a CHAR(2) variable and a BASED FIXED BIN(15) variable for both precision and scale. For example:

```
DCL PRCSNC CHAR(2);
DCL PRCSNN FIXED BIN(15) BASED (ADDR(P RCSNC));
DCL SCALEC CHAR(2);
DCL SCALEN FIXED BIN(15) BASED (ADDR(SCALEC));
```

The SQLDAX fields for a precision of 5 and scale of 2 would be:

```
PRCSNN = 5;
SCALEN = 2;
SQLDAPTR->SQLPRCSN = SUBSTR(P RCSNC,2,1);
SQLDAPTR->SQLSCALE = SUBSTR(SCALEC,2,1);
```

Format 2, although more complex, allows PL/I manipulation of the precision and scale fields. For example, the value of the SQLPRCSN field can be determined simply by reversing the substring operation above. That is:

```
SUBSTR(P RCSNC,2,1) = SQLDAPTR->SQLPRCSN;
```

Such an operation cannot be done using format 1.

Because the PL/I SQLDA is declared as a based structure, your program can dynamically allocate an SQLDA of adequate size for use with each EXECUTE statement. For example, the code fragment below allocates an SQLDA adequate for five fields and uses it in an EXECUTE of statement S3:

```
SQLSIZE=5;
ALLOCATE SQLDA SET(SQLDAPTR);
/* Add code to set values and pointers in the SQLDA */
EXEC SQL EXECUTE S3 USING DESCRIPTOR SQLDA;
```

The statement SQLSIZE=5 determines the size of the SQLDA to be allocated by means of the PL/I REFER feature. The ALLOCATE statement allocates an SQLDA of the size desired, and sets SQLDAPTR to point to it. (Before an EXECUTE statement is issued using this SQLDA, your program must fill in its contents.)

You can use a similar technique to allocate an SQLDA for use with a DESCRIBE statement. The following program fragment illustrates the use of SQLDA with DESCRIBE for three fields and a "prepared" statement S1:

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
SQLSIZE = 3;
ALLOCATE SQLDA SET(SQLDAPTR);
EXEC SQL DESCRIBE S1 INTO SQLDA;
IF SQLD > SQLN THEN
    - get a bigger one;
Set SQLDATA and SQLIND;
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;
```

Defining DB2 Server for VSE Data Types for PL/I

Figure 116 (Page 1 of 2). Data Types for PL/I

Description	DB2 Server for VSE Keyword	Equivalent PL/I Declaration
A binary integer of 31 bits, plus sign.	INTEGER or INT	BINARY FIXED(31)
A binary integer of 15 bits, plus sign.	SMALLINT	BINARY FIXED(15)
A packed decimal number, precision p , scale s ($1 \leq p \leq 31$ and $0 \leq s \leq p$). In storage the number occupies a maximum of 16 bytes. Precision is the total number of digits. Scale is the number of those digits that are to the right of the decimal point.	DECIMAL[(p , s)] or DEC[(p , s)](1) ¹	FIXED DECIMAL(p , s)
A single-precision (4- byte) floating-point number, in short System/390 floating-point format.	REAL or FLOAT(p), $1 \leq p \leq 21$	BINARY FLOAT(p) or FLOAT BINARY(p), $1 \leq p \leq 21$ DECIMAL FLOAT(p) or FLOAT DECIMAL(p), $1 \leq p \leq 7$
A double-precision (8- byte) floating-point number, in long System/390 floating-point format.	FLOAT or FLOAT(p), $22 \leq p \leq 53$ or DOUBLE PRECISION	BINARY FLOAT(p) or FLOAT BINARY(p), $22 \leq p \leq 53$ DECIMAL FLOAT(p) or FLOAT DECIMAL(p), $8 \leq p \leq 16$
A fixed-length character string of length n where $0 < n \leq 254$.	CHARACTER[(n)] or CHAR[(n)]	CHARACTER(n)
A varying-length character string of maximum length n . If $n > 254$ or ≤ 32767 , this data type is considered a long field. See "Using Long Strings" on page 45 for more information.	VARCHAR(n)	CHARACTER(n) VARYING
A varying-length character string of maximum length 32,767 bytes.	LONG VARCHAR	CHARACTER(n) VARYING
A fixed-length string of n DBCS characters where $0 < n \leq 127$.	GRAPHIC[(n)]	GRAPHIC(n)
A varying-length string of n DBCS characters. If $n > 127$ or ≤ 16383 , this data type is considered a long field. See "Using Long Strings" on page 45 for more information.	VARGRAPHIC(n)	GRAPHIC(n) VARYING
A varying-length string of DBCS characters of maximum length 16383.	LONG VARGRAPHIC	GRAPHIC(n) VARYING

Figure 116 (Page 2 of 2). Data Types for PL/I

Description	DB2 Server for VSE Keyword	Equivalent PL/I Declaration
A fixed or varying-length character string representing a date. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	DATE	CHARACTER(<i>n</i>) or CHARACTER(<i>n</i>) VARYING
A fixed or varying-length character string representing a time. The minimum and maximum lengths vary with both the format used and whether it is an input or output operation. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIME	CHARACTER(<i>n</i>) or CHARACTER(<i>n</i>) VARYING
A fixed or varying-length character string representing a timestamp. The lengths can vary on input and output. See the <i>DB2 Server for VSE & VM SQL Reference</i> manual for more information.	TIMESTAMP	CHARACTER(<i>n</i>) or CHARACTER(<i>n</i>) VARYING

Notes:

1. NUMERIC is a synonym for DECIMAL and may be used when creating or altering tables. In such cases, however, the CREATE or ALTER function will establish the column (or columns) as DECIMAL.
2. The data type can be stated in any way that is acceptable to PL/I; BIN FIXED(31), BINARY FIXED(31), and FIXED BIN(31) are all equivalent. If several variables have exactly the same attributes, you can combine them in a single DCL statement:

```
DCL (X,Y,Z) BIN FIXED;
```

Using Stored Procedures

The following example shows how to define the parameters in a stored procedure that uses the GENERAL linkage convention. The NOEXECOPS procedure option must be specified.

```
PLISAMP: PROC(PARM1, PARM2, ...)
          OPTIONS(MAIN, NOEXECOPS);

          DCL PARM1 ...          /* first parameter */
          DCL PARM2 ...          /* second parameter */
          .
          .
          .
```

Figure 117. Stored Procedure - Using GENERAL Linkage Convention

The following example shows how to define the parameters in a stored procedure that uses the GENERAL WITH NULLS linkage convention.

```
PLISAMP: PROC(PARM1, PARM2, INDSTRUC)
  OPTIONS(MAIN, NOEXECOPS);

  DCL PARM1 ...      /* first parameter */
  DCL PARM2 ...      /* second parameter */

  DCL 01 INDSTRUC,
      02 IND1 BIN FIXED(15), /* first ind var */
      02 IND2 BIN FIXED(15); /* second ind var */
  .
  .
  .
```

Figure 118. Stored Procedure - Using GENERAL WITH NULLS Linkage Convention

Appendix F. Decision Tables to Grant Privileges on Packages

Contents

How to Use the Decision Tables	340
Decision Tables	341

How to Use the Decision Tables

The DB2 Server for VSE product uses decision tables to determine whether the owner of a package has the authority or the privilege to execute a given statement. There are three possible scores for each static statement:

- 'G' Means that the package owner has the necessary authorization or privilege for this statement such that the owner can receive the RUN privilege.
- 'Y' Means that the package owner has the necessary authorization or privilege for this statement such that the owner can receive the RUN privilege, but not the GRANT option on that privilege.
- 'D' Means that the package owner must have DBA authority to execute the program containing this statement. No entry is made in the authorization catalog tables.

'G' is the highest score, followed by 'Y', followed by 'D'. For example, suppose a program contains three statements. The package owner receives a 'G', on two of them, but a 'Y' on the third (this occurs when the object referenced in the statement does not exist, or the privileges of the object cannot be resolved). In this situation, the database manager assigns the package a 'Y' (the lower score), allowing the owner to run the package but not to grant the RUN privilege on the package to another user. Because the preprocessor does not distinguish between certain SQL statements that are applied to one application server or to another, you can compensate by doing one of the following:

- Use dynamic statements that cause RUNAUTH=G on both application servers.
- Create separate packages on each application server. These separate packages can then be invoked by a mainline program.
- Create dummy tables that have the same user IDs and table names on the other application server.

Dynamic statements are always given a score of 'G'.

The next few pages show tables. In these tables:

'G', 'Y', and 'D' have the meanings outlined above.

'(G)' and '(Y)' mean that the score for the statement is either 'G' or 'Y', an error message is produced when the program is preprocessed, a partial section for the statement is placed into the package, and the authority for the statement is checked again at the time the package is run.

'n/a' means 'not applicable'.

package owner is the authorization ID of the person who preprocesses the program.

Decision Tables

ACQUIRE DBSPACE

		Dbospace Owner		
		1	2	3
Pkg Owner's Authority		PUBLIC	PRIVATE	
			Dbospace Owner is Pkg Owner	Dbospace Owner not Pkg Owner
A	DBA	D	G	D
B	RESOURCE	(G)	G	(G)
C	None of the above	(G)	(G)	(G)

For cases A2 and B2, the system makes an entry in the SYSUSERAUTH catalog table with RESOURCEAUTH set to 'Y'. In addition, the NAME column is set to the package_id and the AUTHOR column is set to the authorization ID of the person who preprocessed the program. The entry indicates the program's dependency.

ALTER DBSPACE

		Dbospace Owner	
		1	2
Pkg Owner's Authority		Dbospace Owner is Pkg Owner	Dbospace Owner not Pkg Owner
A	DBA	G	D
B	non-DBA	G	(G)

ALTER TABLE

		Table Owner		
Pkg Owner's Authority and Tbl Privilege		1 Table Owner is Pkg Owner	2 Table Owner not Pkg Owner	3 Table does not yet exist
A	DBA, no ALTER	n/a	D	D
B	ALTER without GRANT	n/a	Y	n/a
C	ALTER with GRANT	G	G	n/a
D	non-DBA , no ALTER	n/a	(G)	(G)

For cases B2, C1, and C2, the system makes entries in the SYSTABAUTH catalog table with the ALTERAUTH columns set to 'Y'. The entries represent this package's dependency on ALTER privilege for the table.

The preprocessor determines which level of RUN privilege to give the owner. For some SQL statements, privileges are not checked for all objects affected by the statement. For example, when manipulating primary and foreign keys with the ALTER TABLE statement, ALTER privilege is only checked for the table_name following the ALTER TABLE statement rather than all the tables involved. Additional ALTER and REFERENCES privileges are checked at run time.

COMMENT ON

		Table/View Owner	
Pkg Owner's Authority		1 Table/View Owner is Pkg Owner	2 Table/View Owner not Pkg Owner
A	DBA	G	D
B	non-DBA	G	(G)

CREATE INDEX

		Table on which INDEX is based							
		1		2		3		4	
		Table Exists and the Table's Owner				Table does not yet exist and the Table's Owner			
		is Pkg Owner		not Pkg Owner		is Pkg Owner		not Pkg Owner	
A	DBA, no INDEX	n/a		D		(G)		(Y)	
B	non-DBA, INDEX without GRANT	n/a		Y		n/a		n/a	
C	non-DBA, INDEX with GRANT	G		G		n/a		n/a	
D	non-DBA, no INDEX	n/a		(G)		(G)		(Y)	

For cases B2, C1, and C2, the system makes an entry in the SYSTABAUTH catalog table with the INDEXAUTH column set to 'Y'. The entries represent this package's dependency on INDEX authority privilege for the table.

Note: It is possible for the owner of a table to create an index on that table in the name of another authorization ID. This is true even if the table owner does not have DBA authority.

CREATE TABLE

		Table Owner	
		1	2
		Table Owner is Pkg Owner	Table Owner not Pkg Owner
A	DBA	G	D
B	non-DBA	G	(G)

DELETE

There are two decision tables that apply to DELETE:

The Table Where the Deletion Is Applied :

Pkg Owner's Authority and Tbl Privilege		Table/View on which DELETE is applied							
		1		2		3		4	
		Table/View Exists and the Table/View's Owner		Table/View does not yet exist and the Table/View's Owner		is Pkg Owner		not Pkg Owner	
		is Pkg Owner	not Pkg Owner	is Pkg Owner	not Pkg Owner				
A	DBA, no DELETE	n/a	D	(G)	(Y)				
B	non-DBA, DELETE without GRANT	n/a	Y	n/a	n/a				
C	non-DBA, DELETE with GRANT	G	G	n/a	n/a				
D	non-DBA , no DELETE	n/a	(G)	(G)	(Y)				

In cases B2, C1, and C2, the application server makes entries in the SYSTABAUTH catalog table with the DELETEAUTH column set to 'Y'. The entries represent this package's dependency on the DELETE privilege for the table.

Any Tables Referenced in a WHERE Clause :

Note: The authorization checking in the previous decision table precedes the logic of this table. If the first decision table yields a negative SQLCODE, processing stops. Otherwise, the system applies the lowest level of authorization gained from the two decision tables.

Pkg Owner's Authority and Tbl Privilege		Table/Views in WHERE clause							
		1		2		3		4	
		Table/View Exists and the Table/View's Owner		Table/View does not yet exist and the Table/View's Owner		is Pkg Owner		not Pkg Owner	
		is Pkg Owner	not Pkg Owner	is Pkg Owner	not Pkg Owner				
A	DBA, no SELETE	n/a	Y	(G)	(Y)				
B	non-DBA, SELECT without GRANT	n/a	Y	n/a	n/a				
C	non-DBA, SELECT with GRANT	G	G	n/a	n/a				
D	non-DBA , no SELECT	n/a	(G)	(G)	(Y)				

Figure 119. Tables/Views in WHERE clause

In cases B2, C1, and C2, the application server makes entries in the SYSTABAUTH catalog table with the SELECTAUTH column set to 'Y'. The entries represent this package's dependency on SELECT privilege for the table.

In case A2, the system makes an entry in the SYSUSERAUTH catalog table to show this package's dependency on DBA authority.

GRANT for Authorities Statement

		Authority Granted			
		1	2	3	4
		DBA	RESOURCE	CONNECT to another user	CONNECT to self
A	DBA	D	D	D	G
B	Non-DBA	G	G	G	G

INSERT:

There are two decision tables that apply to INSERT:

The Table Where the Insertion Is Applied :

		Table/Views to which INSERT is applied			
		1	2	3	4
		Table/View Exists and the Table/View's Owner		Table/View does not yet exist and the Table/View's Owner	
		is Pkg Owner	not Pkg Owner	is Pkg Owner	not Pkg Owner
A	DBA, no INSERT	n/a	D	(G)	(Y)
B	non-DBA, INSERT without GRANT	n/a	Y	n/a	n/a
C	non-DBA, INSERT with GRANT	G	G	n/a	n/a
D	non-DBA, no INSERT	n/a	(G)	(G)	(Y)

In cases B2, C1, and C2, the system makes entries in the SYSTABAUTH catalog table with the INSERTAUTH column set to 'Y'. The entries represent this package's dependency on INSERT privilege for the table.

Any Tables Referenced in a WHERE Clause of a Subselect :

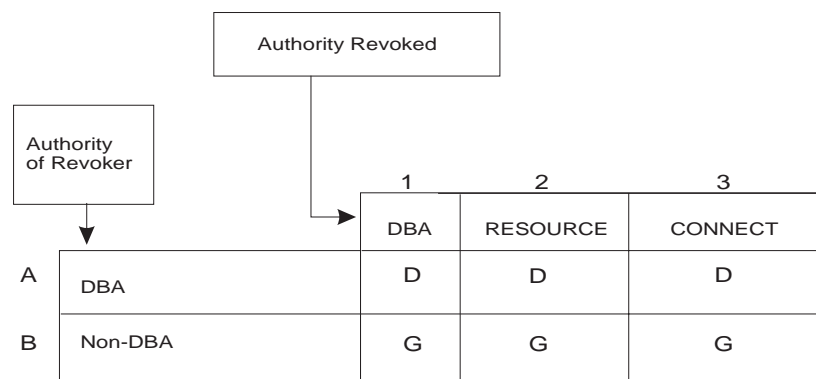
Note: The authorization checking in the previous decision table precedes the logic of this table.

The decision table used here is the same as that used by tables in the WHERE clause of a DELETE in Figure 119 on page 344.

In cases B2, C1, and C2, the system makes entries in the SYSTABAUTH catalog table with the SELECTAUTH column set to 'Y'. The entries represent this package's dependency on SELECT privilege for the table.

In case A2, the system makes an entry in the SYSUSERAUTH catalog table to show this package's dependency on DBA authority.

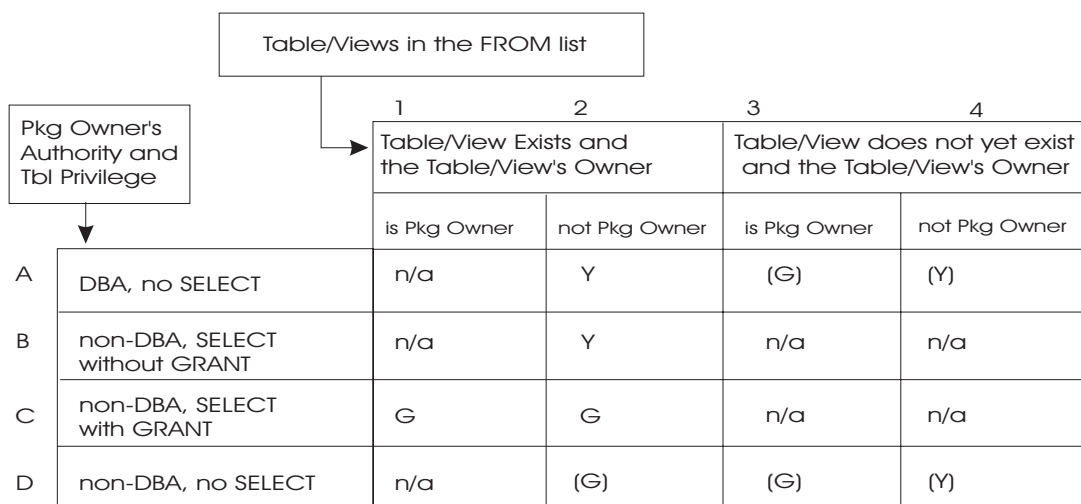
REVOKE for Authorities Statement



SELECT

There are two decision tables that apply to SELECT:

The Tables in the FROM List :



In cases B2, C1, and C2, the application server makes entries in the SYSTABAUTH catalog table with the SELECTAUTH column set to 'Y'. The entries represent this package's dependency on INSERT privilege for the table.

In case A2, there are some instances where a 'Y' entry is made in the DBAAUTH column of the SYSUSERAUTH catalog table, showing package dependencies on DBA authority.

Any Tables Referenced in a WHERE Clause :

Note: The authorization checking in the previous decision table precedes the logic of this table.

The decision table used here is the same as that used by tables in the WHERE clause of a DELETE in Figure 119 on page 344.

In cases B2, C1, and C2, the system makes entries in the SYSTABAUTH catalog table with the SELECTAUTH column set to 'Y'. The entries represent this package's dependency on SELECT privilege for the table.

In case A2, the system makes an entry in the SYSUSERAUTH catalog table to show this package's dependency on DBA authority.

The UPDATE Tables

There are two decision tables that apply to UPDATE:

The Table Where the Update Is Applied :

		Table/Views to which UPDATE is applied			
		1	2	3	4
Pkg Owner's Authority and Tbl Privilege		Table/View Exists and the Table/View's Owner		Table/View does not yet exist and the Table/View's Owner	
		is Pkg Owner	not Pkg Owner	is Pkg Owner	not Pkg Owner
A	DBA, no UPDATE	n/a	D	(G)	(Y)
B	non-DBA, UPDATE without GRANT	n/a	Y	n/a	n/a
C	non-DBA, UPDATE with GRANT	G	G	n/a	n/a
D	non-DBA, no UPDATE	n/a	(G)	(G)	(Y)

In cases B2, C1, and C2, the system makes entries in the SYSTABAUTH catalog table with the UPDATEAUTH column set to 'Y'. The entries represent this package's dependency on UPDATE privilege for the table.

Any Tables Referenced in a WHERE Clause :

Note: The authorization checking in the previous decision table precedes the logic of this table.

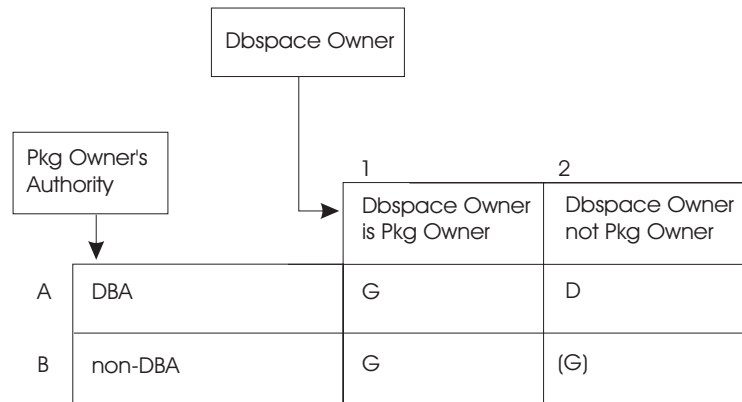
The decision table used here is the same as that used by tables in the WHERE clause of a DELETE in Figure 119 on page 344.

In cases B2, C1, and C2, the system makes entries in the SYSTABAUTH catalog table with the SELECTAUTH column set to 'Y'. The entries represent this package's dependency on SELECT privilege for the table.

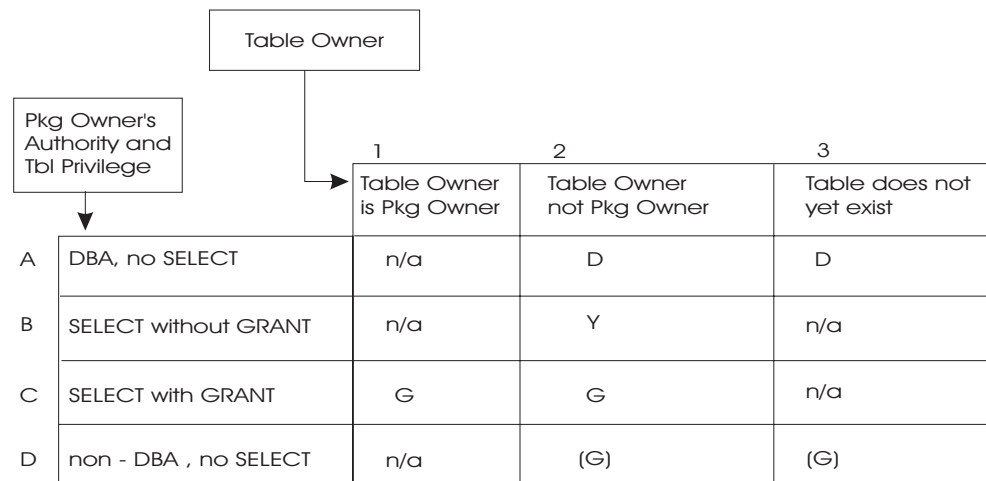
In case A2, the system makes an entry in the SYSUSERAUTH catalog table to show this package's dependency on DBA authority.

There are two decision tables that apply to UPDATE:

The LOCK DBSPACE Table



The LOCK TABLE Table



For cases B1, B2, and C2, the system makes entries in the SYSTABAUTH catalog table. The entries have the SELECTAUTH column set to 'Y' to show the package's dependency.

Bibliography

This bibliography lists publications that are referenced in this manual or that may be helpful.

DB2 Server for VSE Publications

- *DB2 Server for VSE Application Programming*, SC09-2662
- *DB2 Server for VSE Database Administration*, SC09-2655
- *DB2 Server for VSE & VM Database Services Utility*, SC09-2663
- *DB2 Server for VSE Diagnosis Guide and Reference*, LC09-2673
- *DB2 Server for VSE & VM Overview*, GC09-2806
- *DB2 Server for VSE Installation*, GC09-2656
- *DB2 Server for VSE & VM Interactive SQL Guide and Reference*, SC09-2674
- *DB2 Server for VSE Master Index and Glossary*, SC09-2667
- *DB2 Server for VSE Messages and Codes*, GC09-2665
- *DB2 Server for VSE & VM Operation*, SC09-2668
- *DB2 Server for VSE System Administration*, SC09-2658
- *DB2 Server for VSE & VM Performance Tuning Handbook*, GC09-2669
- *DB2 Server for VSE & VM SQL Reference*, SC09-2671

Related Publications

- *DB2 Server for VSE & VM Data Restore*, SC09-2677
- *DRDA: Every Manager's Guide*, GC26-3195
- *IBM SQL Reference, Version 2, Volume 1*, SC26-8416
- *IBM SQL Reference*, SC26-8415

Virtual Storage Extended/Enterprise Systems Architecture (VSE/ESA) Publications

- *IBM VSE/ESA Administration*, SC33-6505
- *IBM VSE/ESA Diagnosis Tools*, SC33-6514
- *IBM VSE/ESA General Information*, GC33-6501
- *IBM VSE/ESA Guide for Solving Problems*, SC33-6510
- *IBM VSE/ESA Guide to System Functions*, SC33-6511

- *IBM VSE/ESA Installation*, SC33-6504
- *IBM VSE/ESA Messages & Codes*, SC33-6507
- *IBM VSE/ESA Networking Support*, SC33-6508
- *IBM VSE/ESA Operation*, SC33-6506
- *IBM VSE/ESA Planning*, SC33-6503
- *IBM VSE/ESA System Control Statements*, SC33-6513
- *IBM VSE/ESA System Macros User's Guide*, SC33-6515
- *IBM VSE/ESA System Macros Reference*, SC33-6516
- *IBM VSE/ESA System Utilities*, SC33-6517
- *IBM VSE/ESA Unattended Node Support*, SC33-6512
- *IBM VSE/ESA Using IBM Workstations*, SC33-6509

CICS/VSE Publications

- *CICS/VSE Application Programming Reference*, SC33-0713
- *CICS/VSE Application Programming Guide*, SC33-0712
- *CICS Application Programming Primer (VS COBOL II)*, SC33-0674
- *CICS/VSE CICS-Supplied Transactions*, SC33-0710
- *CICS/VSE Customization Guide*, SC33-0707
- *CICS/VSE Facilities and Planning Guide*, SC33-0718
- *CICS/VSE Intercommunication Guide*, SC33-0701
- *CICS/VSE Performance Guide*, SC33-0703
- *CICS/VSE Problem Determination Guide*, SC33-0716
- *CICS/VSE Recovery and Restart Guide*, SC33-0702
- *CICS/VSE Release Guide*, GC33-0700
- *CICS/VSE Report Controller User's Guide*, SC33-0705
- *CICS/VSE Resource Definition (Macro)*, SC33-0709
- *CICS/VSE Resource Definition (Online)*, SC33-0708
- *CICS/VSE System Definition and Operations Guide*, SC33-0706
- *CICS/VSE System Programming Reference*, SC33-0711
- *CICS/VSE User's Handbook*, SX33-6079
- *CICS/VSE XRF Guide*, SC33-0704

CICS/ESA Publications

- *CICS/ESA General Information*, GC33-0155

VSE/Virtual Storage Access Method (VSE/VSAM) Publications

- *VSE/VSAM Commands and Macros*, SC33-6532
- *VSE/VSAM Introduction*, GC33-6531
- *VSE/VSAM Messages and Codes*, SC24-5146
- *VSE/VSAM Programmer's Reference*, SC33-6535

VSE/Interactive Computing and Control Facility (VSE/ICCF) Publications

- *VSE/ICCF Administration and Operation*, SC33-6562
- *VSE/ICCF Primer*, SC33-6561
- *VSE/ICCF User's Guide*, SC33-6563

VSE/POWER Publications

- *VSE/POWER Administration and Operation*, SC33-6571
- *VSE/POWER Application Programming*, SC33-6574
- *VSE/POWER Installation and Operations Guide*, SH12-5329
- *VSE/POWER Networking*, SC33-6573
- *VSE/POWER Remote Job Entry*, SC33-6572

Distributed Relational Database Architecture (DRDA) Library

- *Application Programming Guide*, SC26-4773
- *Architecture Reference*, SC26-4651
- *Connectivity Guide*, SC26-4783
- *DRDA: Every Manager's Guide*, GC26-3195
- *Planning for Distributed Relational Database*, SC26-4650
- *Problem Determination Guide*, SC26-4782

C/370 for VSE Publications

- *IBM C/370 General Information*, GC09-1386
- *IBM C/370 Programming Guide for VSE*, SC09-1399
- *IBM C/370 Installation and Customization Guide for VSE* GC09-1417
- *IBM C/370 Reference Summary for VSE*, SX09-1246
- *IBM C/370 Diagnosis Guide and Reference for VSE* LY09-1805

VSE/REXX Publication

- *VSE/REXX Reference*, SC33-6642

Other Distributed Data Publications

- *IBM Distributed Data Management (DDM) Architecture, Architecture Reference, Level 3*, SC21-9526
- *IBM Distributed Data Management (DDM) Architecture, Implementation Programmer's Guide*, SC21-9529
- *VM/Directory Maintenance Licensed Program Operation and User Guide Release 4*, SC23-0437
- *IBM Distributed Relational Database Architecture Reference*, SC26-4651
- *IBM Systems Network Architecture, Format and Protocol*
- *SNA LU 6.2 Reference: Peer Protocols*
- *Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM Systems Network Architecture, Logical Unit 6.2 Reference: Peer Protocols*
- *Distributed Data Management (DDM) List of Terms*
- *IBM Distributed Data Management (DDM) Architecture, Architecture Reference, Level 3*, SC21-9526
- *IBM Distributed Data Management (DDM) Architecture, Architecture Reference, Level 3*, SC21-9526
- *IBM Distributed Data Management (DDM) Architecture, Architecture Reference, Level 3*, SC21-9526

CCSID Publications

- *Character Data Representation Architecture, Executive Overview*, GC09-2207
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

C/370 Publications

- *IBM C/370 Installation and Customization Guide*, GC09-1387
- *IBM C/370 Programming Guide*, SC09-1384

Communication Server for OS/2 Publications

- *Up and Running!*, GC31-8189
- *Network Administration and Subsystem Management Guide* SC31-8181
- *Command Reference*, SC31-8183
- *Message Reference*, SC31-8185
- *Problem Determination Guide*, SC31-8186

Distributed Database Connection Services (DDCS) Publications

- *DDCS User's Guide for Common Servers*, S20H-4793
- *DDCS for OS/2 Installation and Configuration Guide* S20H-4795

VTAM Publications

- *VTAM Messages and Codes*, SC31-6493
- *VTAM Network Implementation Guide*, SC31-6494
- *VTAM Operation*, SC31-6495
- *VTAM Programming*, SC31-6496
- *VTAM Programming for LU 6.2*, SC31-6497
- *VTAM Resource Definition Reference*, SC31-6498
- *VTAM Resource Definition Samples*, SC31-6499

CSP/AD and CSP/AE Publications

- *Developing Applications*, SH20-6435
- *CSP/AD and CSP/AE Installation Planning Guide*, GH20-6764
- *Administering CSP/AD and CSP/AE on VM*, SH20-6766
- *Administering CSP/AD and CSP/AE on VSE*, SH20-6767
- *CSP/AD and CSP/AE Planning*, SH20-6770
- *Cross System Product General Information*, GH23-0500

Query Management Facility (QMF) Publications

- *QMF General Information*, GC26-4713
- *QMF VSE/ESA Setup and Usage Guide*, GG24-4196
- *Managing QMF for VSE/ESA*, SC26-3252
- *Installing QMF on VSE/ESA*, SC26-3254
- *QMF Learner's Guide*, SC26-4714
- *QMF Advanced User's Guide*, SC26-4715
- *QMF Reference*, SC26-4716
- *Installing QMF on VM*, SC26-4718
- *QMF Application Development Guide*, SC26-4722
- *QMF Messages and Codes*, SC26-4834
- *Using QMF*, SC26-8078

- *Managing QMF for VM/ESA*, SC26-8219

DL/I DOS/VS Publications

- *DL/I DOS/VS Application Programming*, SH24-5009

COBOL Publications

- *VS COBOL II Migration Guide for VSE*, GC26-3150
- *VS COBOL II Migration Guide for MVS and CMS*, GC26-3151
- *VS COBOL II General Information*, GC26-4042
- *VS COBOL II Language Reference*, GC26-4047
- *VS COBOL II Application Programming Guide*, SC26-4045
- *VS COBOL II Application Programming Debugging*, SC26-4049
- *VS COBOL II Installation and Customization for CMS* SC26-4213
- *VS COBOL II Installation and Customization for VSE* SC26-4696
- *VS COBOL II Application Programming Guide for VSE* SC26-4697

Systems Network Architecture (SNA) Publications

- *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084
- *SNA Format and Protocol Reference: Architecture Logic for LU Type 6.2*, SC30-3269
- *SNA LU 6.2 Reference: Peer Protocols*, SC31-6808
- *SNA Synch Point Services Architecture Reference* SC31-8134

Miscellaneous Publications

- *IBM 3990 Storage Control Planning, Installation, and Storage Administration Guide*, GA32-0100
- *Dictionary of Computing*, ZC20-1699
- *APL2 Programming: Using Structured Query Language*, SH21-1056
- *ESA/390 Principles of Operation*, SA22-7201

Related Feature Publications

- *Control Center Installation and Operations Guide for VSE*, GC09-2679
- *IBM Replication Guide and Reference*, S95H-0999

Index

Special Characters

- <
 - See *also* shift-out character convention xvii
- >
 - See *also* shift-in character convention xvii

A

- access**
 - concurrent 204
 - table belonging to other users 29
- adding**
 - columns
 - to a table 208
 - in SQL expressions 52
- additional predicates 42**
- ALL**
 - select-clause 28
- ALL keyword**
 - subqueries 80
- ALLOCATE statement of PL/I 335**
- ALTER TABLE**
 - privileges 236
- altering**
 - table 208
- AND operator 40**
- ANSI 122**
- ANY 80**
- APOST preprocessor parameter 115**
- application epilog**
 - batch applications 12
 - CICS/VSE applications 13
 - ICCF applications 12
- application program**
 - example 17
- ARIDDFP 133**
- ARIDSQLA 133**
- ARIPADR4 133**
- ARIPRDID**
 - DB2 Server for VSE linkage module 133
- ARIPRPA preprocessor 109**
- ARIPRPC preprocessor 109**
- ARIPRPF preprocessor 109**
- ARIPRPP preprocessor 109**
- ARIRRTED, DB2 Server for VSE linkage module 133**
- ARIS6ASD**
 - sample program 250
 - source code 250

- ARIS6CBD**
 - sample program 294
 - source code 294
- ARIS6CD**
 - sample program 270
 - source code 270
- ARIS6FTD 312**
 - sample program 312
 - source code 312
- ARIS6PLD**
 - sample program 326
 - source code 326
- ARISYSDD 133**
- arithmetic error**
 - outer select 225
- arithmetic operator**
 - in syntax diagrams xiv
- assembler**
 - acquiring the SQLDSECT area 250
 - data types 258
 - declaring host variables 253
 - declaring the SQLCA 256
 - declaring the SQLDA 257
 - embedding SQL statements
 - example 7, 255
 - sample program 250
 - stored procedures 265
- atomic integrity 231**
- authority**
 - granting 214
 - overview 215
 - revoking from others 216
- authorization-ID**
 - naming conventions 24
- automatic**
 - revocation of privileges 216
- automatic rollback**
 - data definition statements 207
 - deadlocks 205

B

- backing out**
 - changes 16
- backout**
 - definition 205
- backslash**
 - hex value 271
- based structures 165**
- basic form**
 - description 3

batch applications, modules required 133
batch program termination 12
BEGIN DECLARE SECTION 6
BIND preprocessor parameter 115
blocking 128
blocks
 pages 202

C

C programming language

C 287
C compiler preprocessor 272
case sensitivity 271
character constants in SQL statements 272
considerations 270
data conversion considerations 280
data types 287
date calculations 281
DBCS data 282
declaring host variables 272
declaring SQLCA 285
declaring SQLDA 286
delimiting SQL statements 271
dynamic allocation of SQLDA 286
embedding SQL statements
 example 7
INCLUDE statement 272
NUL-terminated strings and truncation 281
placement of SQL statements 271
preprocessor generated statements 282
reentrant programs 289
stored procedures 289
trigraphs 282
case sensitivity
 C 271
catalog tables 219
CBND transaction 140
CCSIDMixed preprocessor parameter 116
CCSIDsbcs preprocessor parameter 116
CESN transaction 107
changing
 data type of a column 208
CHAR data type
 assembler 258
 COBOL 305
 FORTRAN 321
 PL/I 336
character
 data 57
 string
 constant 57
character constant
 SQL statements
 C 272

character subtype

CCSID 46
constant 57
default values 46
overriding defaults 47
CHARSUB preprocessor parameter 116
CHECK preprocessor parameter 116
checking
 SQLCA 151
choose
 in syntax diagrams xiv
CICS 11
 See also CICS/VSE
CICS/VSE
 CONNECT considerations 11, 107
 DB2 Server for VSE linkage module 133
 logical unit of work considerations 15
 program termination 13
 table entries required 15, 138
 use of WHENEVER 154
CICS/VSE default rules
 user IDs 107
CICSUSER default user ID 107
CIRB transaction 107
clause
 order 74
CLOSE
 format 36
closed state of cursor 32
COB2 preprocessor parameter 117, 300
COBOL
 COB2 compiler option consideration 300
 COBRC compiler option consideration 300
 column 7 302
 continuation of SQL statements 294
 COPYBOOKs 300
 data conversion considerations 301
 data types 305
 DBCS data 302
 declaring host variables 296
 declaring SQLCA 302
 delimiting SQL statements 295
 DYNAM compiler option 308
 dynamic statement restrictions 304
 embedding SQL statements
 example 7
 module required 133
 placement of SQL statements 294
 QUOTE compiler option consideration 300
 sample program 294
 stored procedures 308
 using the COBRC parameter 300
 using the INCLUDE statement 301
COBRC preprocessor parameter 117, 300
colon
 used in indicator variables 59

column
 correlation function 85
 join considerations 67
 naming convention 24
 virtual 62

combining
 queries 91

COMMENT ON 209

COMMIT 15
 recommendations 12

committing changes
 tables 15

common column names 67

comparison
 operator 41

compiling
 program 132

completion code 9, 148
See also SQLCODE

concatenation
 EXECUTE IMMEDIATE 162
 operator 54
 symbol 54

concurrent
 access 204

CONNECT
 application programs 10

CONNECT authority
 description 214
 granting 214

connecting DB2 Server for VSE
 application programs 10

connecting to the application server
 CICS/VSE transactions 107

consistency
 of data 14
 token 117, 134, 139

constant
 CCSID 57
 character subtypes 57
 search conditions 52
 select-list expressions 28

constraint
 defer 242
 enforce 242
 naming conventions 24
 referential 233

contention
 for resources 204

continuation of SQL statements
 assembler 255
 C 271
 COBOL 294
 FORTRAN 312
 PL/I 326
 placement of SQL statements 312

conventions
 syntax diagram notation xiii

conversion of data 178
See also data conversion

converting SQL statements 140

COPYBOOKS, COBOL 300

correlation 81
 function 85
 name 83
 subqueries using joins 86
 subquery 81
 table 83

COUNT(*) in a grouping query 72

CREATE INDEX 208

CREATE SYNONYM 209

CREATE TABLE 207

CREATE VIEW
 WITH CHECK OPTION 62

creating
 indexes 208
 synonym 209
 table 207
 field procedure 207
 NULL value 207
 referential integrity 207
 unique constraint 207
 view 62

CTOKEN preprocessor parameter 117

cultural sorting 31, 226

current row of a cursor 32

cursor
 closed state 32
 closing 36
 current row 32
 declaring 32
 description 32
 fetching 34
 inserting 35
 locking stability 124
 management 32
 name
 syntax rules 33
 naming convention 24
 open state 32
 opening 33
 ordering results 36
 result table 32
 scope 33

D

data
 consistency 14
 inconsistent 14
 virtual 62

- data code**
 - SQLDA
 - datetime 179
- data conversion**
 - C 280
 - COBOL 301
 - dynamically defined statements 178
 - FORTRAN 318
 - join conditions 66
 - PL/I 330
 - summary 48
 - unions 92
- data definition 207**
- data integrity**
 - atomic integrity 231
 - entity integrity 231
 - referential integrity 232
 - unique on columns 232
- Data Propagator Capture**
 - on tables 226
- data type**
 - assembler 258
 - C 287
 - CHARACTER 43
 - COBOL 305
 - DATE 43
 - DECIMAL 43
 - description 6, 43
 - FLOAT 43
 - FORTRAN 321
 - GRAPHIC 43
 - INTEGER 43
 - LONG VARCHAR 43
 - LONG VARGRAPHIC 43
 - PL/I 336
 - SMALLINT 43
 - TIME 43
 - TIMESTAMP 43
 - VARCHAR 43
 - VARGRAPHIC 43
- database administrator (DBA)**
 - description 214
- date arithmetic**
 - C 281
- DATE data type**
 - assembler 258
 - C 289
 - COBOL 307
 - FORTRAN 321
 - PL/I 336
- DB2 Server for VSE**
 - catalog tables 219
 - data types, introduction 6
- DB2 Server for VSE production libraries 106**
- DBA 214**
 - See also* database administrator
- DBA (database administrator)**
 - granting authority 214
- DBCS 147**
 - See also* double-byte character set
- dbextent**
 - description 200
- DBP (Dynamic Backout Program) 15**
- dbspace 24**
 - creating 200
 - defining 200
 - dropping 206
 - lock size 204
 - naming conventions 24
 - owner 200
- deadlock**
 - description 205
- DECIMAL data type**
 - assembler 258
 - C 287
 - COBOL 305
 - FORTRAN 321
 - PL/I 336
- DECLARE CURSOR**
 - coded queries 32
- declaring**
 - host variables
 - assembler 253
 - C 272
 - COBOL 296
 - FORTRAN 313
 - PL/I 327
 - return code structure
 - assembler 256
 - C 285
 - COBOL 302
 - FORTRAN 318
 - PL/I 332
 - static external variables
 - PL/I 327
- default**
 - in syntax diagrams xv
- defining**
 - indexes 208
 - synonym 209
 - tables 207
 - field procedure 207
 - NULL value 207
 - referential integrity 207
 - unique constraint 207
 - view on view 63
- DELETE**
 - delete rules 240
 - rules 240
- delete connected**
 - table 234

- deleting**
 - tables 208
- delimiting SQL statements**
 - assembler 255
 - C 271
 - COBOL 295
 - dynamic execution 162
 - PL/I 326
- dependent**
 - table 234
- descendent**
 - table 234
- DESCRIBE**
 - usage techniques 164
- DFHPCT 15**
- DFHSIT 15**
- DISPLAY SIGN HEADING SEPARATE 301**
- DISTINCT**
 - HAVING clause 74
 - select-clause 27
- distributed applications**
 - application server 2
- distributing packages 139**
- division in SQL expressions 52**
- double precision float 44**
- double quotation marks**
 - considerations in COBOL 115
- double-byte character set (DBCS)**
 - characters for C 287
 - constants 58
 - data 282
 - assembler 256
 - C 282
 - COBOL 302
 - FORTRAN 318
 - PL/I 331
 - data type 307, 336
 - assembler 258
 - FORTRAN 321
- DROP DBSPACE 206**
- DROP TABLE 208**
- DROP VIEW 66**
- dropping**
 - column 208
 - table 208
 - view 66
- DSECTs used by the database manager 250**
- DTB (Dynamic Transaction Backout) 15**
- duration**
 - date 59, 222
 - labeled 222
 - lock 204
 - time 59, 222
 - timestamp 59, 224
- DYNAM compiler option**
 - COBOL 308

- Dynamic Backout Program (DBP) 15**
- dynamic data conversion 178**
 - assembler 257
 - C 286
 - COBOL 304
 - FORTRAN 319
 - PL/I 334
- dynamic SQL**
 - description 24
- dynamic statements**
 - comparison with extended dynamic statements 186
 - data conversion 178
 - description 184
 - extended 184, 193
 - comparison with dynamic statements 186
 - introduction 184
 - relationship between 185
 - logical unit of work considerations 193
 - SQLDA use 179
- Dynamic Transaction Backout (DTB) 15**

E

- END DECLARE SECTION 6**
- ending logical units of work**
 - batch applications 12
 - CICS/VSE transactions 13
 - ICCF applications 12
- entity integrity 231**
- erasing**
 - tables 208
- error code 9, 148**
 - See also* SQLCODE
- error handling**
 - application programs 148
 - description 147
 - preprocessors 109
- ERROR preprocessor parameter 116, 117**
- error recovery 147**
 - See also* error handling
- evaluating**
 - predicates 41
- examining the SQLCA 151**
- exclusive lock**
 - description 204, 205
- EXECUTE**
 - usage techniques 163
- EXECUTE IMMEDIATE**
 - usage techniques 161
- EXECUTE privilege**
 - automatic revocation 216
- EXIST preprocessor parameter 117**
- EXISTS predicate 90**
- expanding tables 208**
- EXPLAIN preprocessor parameter 118**

expression

- adding 52
- constants used 52
- description 51, 53
- dividing 52
- host variables 52
- multiplying 52
- subtracting 52

extended dynamic statement

- assembler example 187
- comparison with dynamic statements 186
- description 184
- introduction 184
- logical unit of work considerations 193
- relationship between 185

F

FETCH

- description 34
- error handling for a select-list 156
- format 34

fetch and insert blocking 128

field procedure

- collating sequence 31
- using 225

FIPS 122

Flagger

- description 122
- SQLFLAG 120

FLOAT data type

- assembler 258
- C 287
- COBOL 305
- FORTTRAN 321
- PL/I 336

FOR update-clause 31

foreign key 234

FORTTRAN 318

- continuation of SQL statements 312
- data types 321
- declaring host variables 313
- embedding SQL statements 315
 - example 9
- long character strings 315
- preprocessor
 - restrictions 320
- sample program 312

fragment of syntax

- in syntax diagrams xvi

FROM

- correlation name 83
- description 29
- joins 66
- PREPARE statement 181

fullselect

- description 26

functions

- used in grouping 71

G

general rules for naming data objects 24

GRANT option

- description 215

granting

- authorities 214

graphic constants

- PL/I formats 58
- SQL format 58

GRAPHIC data type

- assembler 258
- C 287
- COBOL 307
- FORTTRAN 321
- PL/I 336

Graphic preprocessor parameter 118

GROUP BY clause

- correlated subquery considerations 85
- description 71
- subqueries 81

group query 29

- See also* group-by-clause

group-by-clause

- description 29

grouping

- feature 71

grouping queries

- VARCHAR 72
- VARGRAPHIC 72

H

HAVING clause

- correlated subqueries 85
- description 30, 73
- subqueries 78, 81

header pages 202

hexadecimal constants

- within expressions 57

host language

- description 3

host structure 253, 272, 296, 313, 327

- COBOL 299
- description 55

host variable

- assembler 256
- declared in assembler 253
- declared in C 272, 278, 330
- declared in COBOL 296
- declared in FORTTRAN 313

host variable (*continued*)
 declared in PL/I 327
 description 54
 dynamically defined statements 181
 FORTRAN 317
 function parameters
 used in C 280
 in syntax diagrams xiii
 INTO clauses 38
 naming convention 24
 nulls 59
 PL/I 330
 restriction on use in CREATE VIEW 63
 restriction on use in REVOKE 216
 search conditions 52
 truncation 59
 used in a program 54

host-program variable 54
See also host variable

I

ICCF applications, modules required 133
ICCF program termination 12
identifier
 description 24
 long 69

implicit
 connection 107
 revocation of privileges 216

IN predicate 80

inactive
 constraint 242
 removing referential constraints 244
 table 242

INCLUDE 131
 assembler 256
 C 272
 COBOL 301
 FORTRAN 318
 PL/I 327

INCLUDE SQLCA
 assembler language 256
 C 285
 COBOL 302
 FORTRAN 318
 PL/I 332
 pseudocode 9, 148

INCLUDE SQLDA
 assembler language 257
 C 286
 PL/I 334
 pseudocode 165

including
 CICS/VSE procedures 133
 external source members 131

including (*continued*)
 extra linkage modules 133
 input from Source Statement Library 131
 relocatable modules 132

inconsistency
 because of cursor stability 125
 cursor stability inconsistencies 125

inconsistent
 data 14
 state 14, 147

index
 creating 208
 naming convention 24
 restriction for view 61

indicator array
 description 55
 in assembler 253
 in C 272
 in COBOL 296
 in FORTRAN 313
 in PL/I 327

INDICATOR keyword
 indicator variables 59

indicator variable
 description 55, 59
 detecting nulls 61
 detecting truncation 61
 FETCH statement 34
 in assembler 253
 in C 272
 in COBOL 296
 in FORTRAN 313
 in PL/I 327
 meaning of values returned 60

initializing
 SQLDA 179

input host variables 34

INSERT
 INSERT rules 237
 rules for referential integrity 237

insert and fetch blocking 128

INTEGER data type
 assembler 258
 C 287
 COBOL 305
 data types 336
 FORTRAN 321
 PL/I 336

integrity
 atomic 231
 data 232
 entity 231
 referential 232

internal statistics, updating 208

INTO clause
 description 37

INTO clause (*continued*)

- dynamically defined statements 166
- FETCH statement 34
- restriction for subqueries 78
- unions 91

invoking

- PL/I preprocessor
 - multiple user mode 110

invoking the preprocessor 108**ISO 122****isolation level**

- cursor stability 124
- mixing 127
- repeatable read 124
- USER option 118, 127

ISOLation preprocessor parameter 118, 127**J****job control examples**

- multiple user mode preprocessing 110
- suppressing printout 114

join

- common column names 67
- correlated subqueries 86
- data conversion 66
- description 66
- join variable 69
- limits 71
- nulls 68
- number permitted 71
- referring to another user's table 67, 70
- SELECT * 71
- single table (to itself) 69
- trailing blanks 68
- without join conditions 66

K**KEEP parameter 119****keeping authorization names and password secret 114****key**

- foreign 234
- primary 234

keyword

- in syntax diagrams xiii

L**LABEL**

- SQL 209

LABEL preprocessor parameter 119**limits**

- joins 71

link-editing

- extra modules required 133
- module not found 132

loading

- program 132

LOCK 205

- ACQUIRE DBSPACE 203
- ALTER DBSPACE 204

lock size

- definition 201

locking

- dbspaces
 - explicitly 205
 - modifying 201
 - reading 201
- description 204
- duration 204
- exclusive 204
- isolation level
 - considerations 124
- share 204
- tables explicitly 205
- update 204

LOG job control statement 114**logical operator 40****logical unit of work (LUW)**

- automatic locking 204
- automatic rollback 205, 207
- batch considerations 12
- CICS/VSE considerations 13, 15, 107
- committing work done 15
- description 14, 147
- error handling 147
- ICCF considerations 12
- revoking privileges 216
- rolling back work done 16
- using extended dynamic statements 193

long character strings in FORTRAN 315**long strings**

- description 45
- use 45

LONG VARCHAR

- restrictions 45
- unions 93

LONG VARCHAR data type

- assembler 258
- C 287
- COBOL 305
- FORTRAN 321
- PL/I 336

LONG VARGRAPHIC

- restrictions 45
- unions 93

LONG VARGRAPHIC data type

- assembler 258
- C 289

LONG VARGRAPHIC data type *(continued)*

COBOL 307
FORTRAN 321
PL/I 336

M

main variable 55, 253, 272, 296, 313, 327

manipulating a cursor 33

maximum

joins 71

merging results of queries 91

mixing isolation levels 127

modifying

locked dbspace 201

tables through a view 64

module not found 132

multiple

row

query results 32

multiple row results 32

multiple user mode

invoking the preprocessors 108, 110

multiple-partition mode

locking considerations 204

multiplication in SQL expressions 52

N

naming

column 24

data object 24

dbspace 24

index 24

table 24

negative SQLCODE

description 9, 148

nesting correlated subqueries 87

NHEADER

ACQUIRE DBSPACE 202

NOBIND preprocessor parameter 115

NOCHECK preprocessor parameter 116

NOEXIST preprocessor parameter 117

NOFOR preprocessor parameter 118

NOGRaphic preprocessor parameter 118

NOLOG job control statement 114

nonexecutable SQL statements 10

NOPACKAGE preprocessor parameter 120

NOPRint preprocessor parameter 120

NOPUnch preprocessor parameter 120

NOSEQUence preprocessor parameter 120

NOSQLCA

support 121, 148

NOT EXISTS predicate 90

not found SQLCODE (100)

FETCH 34

NOT IN predicate 80

NOT keyword

concatenation 40

NUL-terminated strings and truncation

C 281

null value 88

grouping queries 73

indicator variables 59, 60

joins 68

search conditions 40

NUMERIC

See DECIMAL 45

O

online applications 11

See also CICS/VSE

online environment 11

See also CICS/VSE

online programs 11

See also CICS/VSE

OPEN

description 33

format 33

open state of a cursor 32

operator

arithmetic 52

comparison 41

logical 40

operator ID 107

OPTION LOG job control statement 114

OPTION NOLOG job control statement 114

optional

default parameter

in syntax diagrams xv

item

in syntax diagrams xiv

keyword

in syntax diagrams xv

OPTIONS(MAIN) clause 326

OR operator 40

order

clauses 74

ORDER BY clause

description 36

restriction for CREATE VIEW 63

unions 91, 93

ordering

query results 36

output host variables 34

owner

dbspace 200

P

package

- automatic regeneration 134, 216
- description 108
- distributing 139
- invalidating
 - DROP DBSPACE 206
 - DROP VIEW 66
 - REVOKE 216

PACKAGE preprocessor parameter 120

page

- header 202

PAGE lock size 204

PAGES parameter of ACQUIRE DBSPACE 202

parameter

- marker 162, 181

parameterized statement

- description 162

parent table

- table 233

parentheses

- in syntax diagrams xiv

password

- naming convention 24

PCT (Program Control Table) 15

PCTFREE

- parameter of ACQUIRE dbspace 203
- parameter of ALTER DBSPACE 204

PCTINDEX

- parameter of ACQUIRE DBSPACE 203

performance

- considerations 124

PL/I 336

- attributes of variables 329
- continuation of SQL statements 326
- data conversion considerations 330
- declaring host variables 327
- declaring SQLCA 332
- declaring SQLDA 334
- declaring static external variables 327
- delimiting SQL statements 326
- dynamic allocation of SQLDA 334
- embedding SQL statements
 - example 9
- placement of SQL statements 326
- preprocessing programs coded 110
- sample program 326
- stored procedures 337
- using the INCLUDE statement 327

placement of SQL statements

- assembler 255
- C 271
- COBOL 294
- FORTTRAN 312
- PL/I 326

pointer type attribute

- C 278

positions of a cursor 32

positive SQLCODE

- description 9, 148

potential deadlocks 205

precedence rules 40, 52

predicates

- constants 52
- description 40
- host variables 52
- rules for evaluating 41

PREPARE

- format 181
- usage techniques 163

PrepFile preprocessor parameter 115

PREPNAME preprocessor parameter 114

PREPPARM preprocessor parameter 115

preprocessing

- multiple user mode 108
 - PL/I 110
- option file 121
- programs 109
- single user mode 108

preprocessor parameters 124

- APOST 115
- BIND 115
- CCSIDMixed 116
- CCSIDSbcs 116
- CHARSUB 116
- CHECK 116
- COB2 117, 300
- COBOL 300
- COBRC 117
- CTOKEN 117
- DATE 117
- ERROR 116, 117
- EXIST 117
- EXPLAIN 118
- GRaphic 118
- ISOLation 118, 127
- KEEP 119
- LABEL 119
- LineCount 119
- NOBIND 115
- NOCHECK 116
- NOEXIST 117
- NOFOR 118
- NOGRaphic 118
- NOPACKAGE 120
- NOPRint 120
- NOPUnch 120
- NOSEQUence 120
- PACKAGE 120
- PrepFile 115
- PREPNAME 114

preprocessor parameters *(continued)*

PREPPARM 115
PRint 120
PUnch 120
QUOTE 115, 300
REVOKE 119
SEQuence 120
SQLFLAG 120
STDSQL 121
suppressing display 114
SYSIN 120
TIME 121
USERid 114

primary key 234

PRint preprocessor parameter 120

privilege

automatic revocation 216
description 215
package 216
programs 218
references 236
revoking 218
 from others 216
table or view
 alter 215
 delete 215
 description 215
 index 215
 insert 215
 references 215
 select 215
 update 215

program

interrupts
 FORTRAN 319
 PL/I 333
naming convention 24
privileges
 revoking 218
sample 17
termination
 batch applications 12

Program Control Table (PCT) 15

program termination

CICS/VSE programs 13
ICCF applications 12

programming interface

description ix

programs using DBCS data

assembler 256
C 282
COBOL 302
FORTRAN 318
PL/I 331

PSERVERs

characteristics 211

PSERVERs *(continued)*

example of a definition 212
execution 211

pseudocode

description 3

PUnch preprocessor parameter 120

punctuation mark

in syntax diagrams xiv

PUT

description 35
format 35

Q

qualifiers

column names 67
table names 24

qualify

fully 24

qualifying groups 30

See also having clause

Query

Connected Database 53

querying

tables through a view 63

quotation mark

considerations in COBOL 115
constants 57

QUOTE preprocessor parameter 115, 300

R

reading from a locked dbspace 201

REBIND PACKAGE 135

reentrant programs

assembler 260, 289
C 289

REFER feature of PL/I 335

references privilege 236

existing applications 237
modifying the contents of tables with referential constraints 237

referential

constraint 233
cycle
 definition 233
 description 232
integrity 232
 existing tables 235
 new tables 234
 rules 242
structure 233

register save conventions

support 135

relationship 233

RELEASE option

COMMIT 12
ROLLBACK 12

releasing your connection

batch applications 12
CICS/VSE applications 13
CMS applications 15
ICCF applications 12
to DB2 Server for VSE 15

remote unit of work

extended dynamic processing 187

repeat symbol

in syntax diagrams xv

repeatable read locking 124**required item**

in syntax diagrams xiv

reserved words

SQL xvi

RESOURCE authority

granting 214

restoring data 16**restriction**

unions 92

result code 9, 148

See also SQLCODE

result table

cursor 32

retrieving

columns of a row 28

return code 9, 148

See also SQLCODE

REVOKE 216**REVOKE preprocessor parameter 119****revoking**

privileges 216
automatic 216
overview 216
programs 218

ROLLBACK 12, 16

description 147

ROLLBACK WORK 205**rollback, automatic**

data definition statements 207
deadlocks 205

rolling back changes 16**ROW lock size 204****rules**

evaluating predicates 41
naming data object 24
SQL in assembler 253
SQL in C 270
SQL in FORTRAN 312
SQL in PL/I 326

S**sample program**

ARIS6ASD 250
ARIS6CBD 294
ARIS6CD 270
ARIS6FTD 312
ARIS6PLD 326
list 17

SCHEDULE authority

granting 214

scope

cursor 33
WHENEVER statement 151

search condition 29

See also where-clause

AND operator 40
arithmetic operators 52
comparison operators 41
constant 52
description 38
expressions 53
host variables 52
join conditions 66
NOT keyword 40
OR operator 40
precedence rules 40, 52
predicates 40

SELECT

ALL 28
ALL keyword 80
ANY keyword 80
clause 27
constants 28
correlation 81
DISTINCT keyword 27
error handling 156
EXISTS 90
group-by-clause 29
grouping 71
having-clause 30
INTO clause 37
introduction 25
joins 66
NOT EXISTS predicate 90
order of clauses 74
ordering results 36
SELECT * form 28
select-clause 27
select-list 27
subqueries 77
unions 91
where-clause 29

SELECT *
basic queries 28
join 71

SELECT * (*continued*)
 views 62

select list
 constants 28
 description 27
 error handling 156
 restrictions
 GROUP BY 73

select-statement
 description 26

selecting
 all columns of a row 28
 isolation level 126

self-referencing table 234

sequence of clauses 74

SEQUENCE preprocessor parameter 120

share lock
 description 204, 205

shift-in character
 convention xvii

shift-out character
 convention xvii

single precision float 44

single quotation mark
 COBOL considerations 115, 300
 considerations in COBOL 115
 constant 57

single user mode
 invoking the preprocessors 108

single-partition mode
 locking considerations 204

single-row query results 38

SIZE=AUTO
 invoking preprocessors 110

SMALLINT data type
 assembler 258
 C 287
 COBOL 305
 FORTRAN 321
 PL/I 336

SOME keyword 80

source code
 ARIS6ASD 250
 ARIS6CBD 294
 ARIS6CD 270
 ARIS6FTD 312
 ARIS6PLD 326

Source Statement Library
 input 131

source value 48

special register
 description 53
 expressions 53

special statements
 UPDATE STATISTICS 208

SQL comments
 assembler 255
 C 271
 COBOL 295
 FORTRAN 312
 in static SQL statements 94
 PL/I 326

SQL declare section 6, 54, 55, 253, 272, 296, 313, 327

SQL Descriptor Area (SQLDA)
 assembler declaration 257
 C declaration 286
 COBOL declaration 304
 FORTRAN declaration 319
 PL/I declaration 334
 Structure 179

SQL identifier
 description 24

SQL statements
 embedding in application program
 examples 10

SQL-89
 conformance checking 120, 122, 197

SQLCA (SQL Communications Area)
 assembler declaration 256
 C declaration 285
 COBOL 302
 description 148
 FORTRAN 318
 PL/I declaration 332
 testing 151

SQLCODE 9, 148

SQLCODE 100 (not found)
 FETCH 34

SQLD field in the SQLDA 180

SQLDA (SQL Descriptor Area)
 summary 179

SQLDAX structure (in PL/I) 334

SQLDSECT, acquiring 250

SQLDSIZ variable 250

SQLERRM
 description 149

SQLFLAG preprocessor parameter 120

SQLN
 field in SQLDA 179
 setting 179

SQLWARN
 description 150

SQLWARN6
 automatic rollback 205

starter database 106

statements
 ALTER TABLE 208
 BEGIN DECLARE SECTION 6
 CLOSE 36
 COMMENT ON 209

statements *(continued)*

- COMMIT 15
- CONNECT 10
- CREATE INDEX 208
- CREATE SYNONYM 209
- CREATE TABLE 207
- CREATE VIEW 62
- DECLARE 32
- DROP DBSPACE 206
- DROP SYNONYM 209
- DROP TABLE 208
- DROP VIEW 66
- END DECLARE SECTION 6
- FETCH 34
- INCLUDE 131
- INCLUDE SQLCA 9, 148
- INCLUDE SQLDA 165
- LABEL 209
- LOCK 205
- naming convention 24
- OPEN 33
- PREPARE 181
- PUT 35
- REVOKE 216
- ROLLBACK 16
- SELECT
 - basic use 25
 - correlation 81
 - grouping 71
 - joins 66
 - subqueries 77
 - testing for existence 90
 - unions 91
- UPDATE STATISTICS 208
- WHENEVER 150

static SQL

- description 24

statistics

- tables 208

storage pool

- definition 200
- nonrecoverable 200
- recoverable 200
- specifying the placement of dbspaces 203

stored procedures

- assembler 265
- authorizing 100
- benefits 96
- C 289
- calling 99
- characteristics 209
- COBOL 308
- coding examples 103
- datatype compatibility 101
- example of a definition 210
- execution of SQL statements 98

stored procedures *(continued)*

- GENERAL linkage convention 102
- GENERAL WITH NULLS linkage convention 102
- Language Environment considerations 98
- parameters 100
- passing parameters 101
- PL/I 337
- preparing 98
- result sets 104
- specifying AUTHIDs 100
- using 95, 209
- writing 97

STORPOOL parameter of ACQUIRE DBSPACE 203**structures**

- based 165

subquery

- ALL keyword 80
- ANY keyword 80
- correlation 81
- IN predicate 80
- introduction 77
- involving unions (restriction) 93
- many values returned 80
- NOT IN predicate 80
- single value returned 79

subselect

- description 26

subtraction in SQL expressions 52**subtype**

- character 46
- Using 57

success code 9, 148

- See also* SQLCODE

summary

- program framework 16

suppressing SYSPCH and SYSLST output 109**suppressing the preprocessor parameter display 114****syncpoints 13**

- See also* logical units of work

synonym

- creating 209
- naming convention 24

syntax diagram

- notation conventions xiii

SYSIN preprocessor parameter 120**SYSLST, suppressing preprocessor output 109****SYSPCH, suppressing preprocessor output 109****system initialization table 15****Systems Application Architecture (SAA)**

- conformance checking 120, 122, 197

T**table**

- accessing 29

table (*continued*)
 altering 208
 creating 207
 indexes 208
 synonyms 209
 defining labels 209
 delete-connected 234
 dependent 234
 descendent 234
 designator
 rules 70, 90
 dropping 208
 entering comments in catalog tables 209
 exposed table name 70
 labels 69
 naming convention 24
 nonexposed table name 70
 or view
 privileges 215
 parent table 233
 self-referencing 234
tables, CICS/VSE 15
target value 48
terminal operator id 107
termination
 batch programs 12
 CICS/VSE programs 13
 ICCF applications 12
testing
 existence 90
 SQLCA 151
TIME data type
 assembler 258
 C 289
 COBOL 307
 FORTRAN 322
 PL/I 336
TIMESTAMP
 data type
 assembler 260
 C 289
 COBOL 307
 FORTRAN 322
 PL/I 337
 duration 224
trailing blanks 68, 72
transactions 11
 See also CICS/VSE
trigraphs
 C 282

U

UNION ALL 91
UNION operator
 CCSID 93

UNION operator (*continued*)
 character subtypes 93
 description 91
 ordering results 91, 93
 restriction for CREATE VIEW 63
 usage restrictions involving
 data types 92
 LONG VARCHAR data 93
 subqueries 93
 VARCHAR and VARGRAPHIC 93
 views 93
unions 91
 See also UNION operator
unique on columns 232
uniquely identifying an object 24
unresolved external reference 133
UPDATE
 rules 238
update rule 238
UPDATE STATISTICS 208
updating
 internal statistics 208
user ID
 CICS/VSE default rules 107
 CICSUSER default 107
 naming convention 24
USER special register 54
USERid
 preprocessor parameter 114

V

valid lock size 204
VARCHAR data type
 assembler 258
 C 287
 COBOL 305
 FORTRAN 321
 PL/I 336
VARGRAPHIC
 constants within expressions 58
 data type
 assembler 258
 C 287
 COBOL 307
 FORTRAN 321
 PL/I 336
view
 CREATE VIEW 62
 description 61
 DROP VIEW 66
 materialization 64
 modifying tables through 64
 naming convention 24
 privileges 215
 querying tables through 63

view (*continued*)

- unions
 - restriction 93

virtual

- columns 62
- data 62

VM

- loading a program 132

VSE

- CICS/VSE CONNECT considerations 107
- compiling a program 132
- preprocessing programs 109
- register save conventions 135

W

warning

- conditions 150
- flags 150

WHENEVER 10, 148, 150

WHERE clause

- ALL keyword 80
- ANY keyword 80
- correlated subquery 81
- description 29
- EXISTS predicate 90
- grouping considerations 73
- IN predicate 80
- join conditions 66
- NOT EXISTS predicate 90
- NOT IN predicate 80
- subqueries 77

WITH CHECK OPTION 62

WITH clause 32

writing clauses in order 74

Z

zero SQLCODE

- description 9, 148

Communicating Your Comments to IBM

DB2® Server for VSE
Application Programming
Version 6 Release 1
Publication No. SC09-2662-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
 - United States and Canada: 416-448-6161
 - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
 - Internet: torrcf@ca.ibm.com
 - IBMLink: [toribm\(torrcf\)](mailto:toribm(torrcf)@ca.ibm.com)
 - IBM/PROFS: [torolab4\(torrcf\)](mailto:torolab4(torrcf)@ca.ibm.com)
 - IBMMAIL: [ibmmail\(caibmwt9\)](mailto:ibmmail(caibmwt9)@ca.ibm.com)

Readers' Comments — We'd Like to Hear from You

DB2® Server for VSE
Application Programming
Version 6 Release 1
Publication No. SC09-2662-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? Yes No

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

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape

Please do not staple

Fold and Tape



File Number: S370/4300-50
Program Number: 5648-A70



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2662-00



Spine information:



DB2® Server for VSE

Application Programming

Version 6 Release 1