IBM DB2 10.1
for Linux, UNIX, and Windows

# *Preparation Guide for DB2 10.1 Fundamentals Exam 610*

SC27-4540-00

IBM DB2 10.1
for Linux, UNIX, and Windows

# *Preparation Guide for DB2 10.1 Fundamentals Exam 610*

SC27-4540-00

**Note**

Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 457.

**Edition Notice**

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

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at http://www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide/

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

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.

# Contents

# About this book

This book provides information from the DB2® for Linux, UNIX, and Windows documentation to cover all the objectives that are described in the DB2 10.1 Fundamentals Exam 610.

- Part 1, "DB2 Database for Linux, UNIX, and Windows," on page 1 provides information about DB2 products, editions, and features.
- Part 2, "Security," on page 19 provides information about the DB2 security model, authorization, authorities, privileges, roles, trusted context, and Row and Column Access Control (RCAC).
- Part 3, "Working with Databases and Database Objects," on page 67 provides information about DB2 servers, DB2 databases, database connectivity, database objects, data concepts, data types, and DDL statements to create database objects such as schemas, tables, constraints, views, triggers, and routines.
- Part 4, "Working with DB2 Data using SQL," on page 247 provides information about SQL statements to manage data and retrieve data, including tables with XML columns and temporal tables. Also, it provides information about how to invoke user-defined functions and call procedures.
- Part 5, "Data concurrency," on page 403 provides information about data concurrency, isolation levels, lock characteristics, locks that can be obtained in database objects, and factors that influence locking.

You must pass DB2 10.1 Fundamentals Exam 610 to obtain the *IBM® Certified Database Associate - DB2 10.1 Fundamentals* certification. For complete details about this certification, visit http://www.ibm.com/certify/certs/08003504.shtml.

## Who should use this book

This book is for users of DB2 for Linux, UNIX, and Windows who want to prepare for the certification Exam 610. For complete details about the exam, visit http://www.ibm.com/certify/tests/ovr610.shtml.

# Part 1. DB2 Database for Linux, UNIX, and Windows

DB2 for Linux, UNIX, and Windows is optimized to deliver industry-leading performance across multiple workloads, while lowering administration, storage, development, and server costs.

DB2 for Linux, UNIX, and Windows offers numerous features that help lower the cost of managing data by automating administration, reduce the cost of storage with industry leading data compression technologies, optimize workload execution, deliver scalability and high availability, secure data access, reduce the cost of developing and maintaining applications, and licensing terms for virtualized environments.

DB2 for Linux, UNIX, and Windows offers multiple editions designed to meet the needs of different business environments

**Related information**:

DB2 database product page

# Chapter 1. DB2 database product editions

There are multiple DB2 database product editions, each with a unique combination of features and functionality.

**DB2 Advanced Enterprise Server Edition and DB2 Enterprise Server Edition**
Ideal for high-performing, robust, on-demand enterprise solutions.

DB2 Enterprise Server Edition is designed to meet the data server needs of mid-size to large-size businesses. It can be deployed on Linux, UNIX, or Windows servers of any size, from one processor to hundreds of processors, and from physical to virtual servers. DB2 Enterprise Server Edition is an ideal foundation for building on demand enterprise-wide solutions such as high-performing 24 x 7 available high-volume transaction processing business solutions or Web-based solutions. It is the data server backend of choice for industry-leading ISVs building enterprise solutions such as business intelligence, content management, e-commerce, Enterprise Resource Planning, Customer Relationship Management, or Supply Chain Management. Additionally, DB2 Enterprise Server Edition offers connectivity, compatibility, and integration with other enterprise DB2 and IDS data sources.

Program charges: DB2 Enterprise Server Edition includes table partitioning, high availability disaster recovery (HADR), online reorganization, materialized query tables (MQTs), multidimensional clustering (MDC), query parallelism, connection concentrator, the governor, pureXML®, backup compression, and Tivoli® System Automation for Multiplatforms (SA MP). The product also includes DB2 Homogeneous Federation and Homogeneous SQL Replication allowing federated data access and replication between DB2 servers as well as Web services federation. DB2 Enterprise Server Edition is available on either a Processor Value Unit or per Authorized User pricing model. You must acquire a separate user license for each Authorized User of this product with a minimum purchase of 25 users per 100 Processor Value Units.

For more information, refer to www.ibm.com/software/data/db2/linux-unix-windows/edition-enterprise.html.

**DB2 Workgroup Server Edition**
Ideal for departmental, workgroup, or medium-sized business environments.

DB2 Workgroup Server Edition is the data server of choice for deployment in a departmental, workgroup, or medium-sized business environment. It is offered in per Authorized User, Value Unit, or limited use socket pricing models to provide an attractive price point for medium-size installations while providing a full-function data server.

Program charges: DB2 Workgroup Server Edition includes high availability disaster recovery (HADR), online reorganization, pureXML, Web services federation support, DB2 Homogeneous Federation, Homogeneous SQL Replication, backup compression, and Tivoli System Automation for Multiplatforms (SA MP). DB2 Workgroup Server Edition can be deployed in Linux, UNIX, and Windows server environments and will use up to 16 cores and 64 GB of memory. DB2 Workgroup Server Edition is restricted to a stand-alone physical server with a specified maximum number of Processor Value Units based on the total number and type of processor

cores, as determined in accordance with the IBM Express Middleware™ Licensing Guide available at ftp://ftp.software.ibm.com/software/smb/pdfs/LicensingGuide.pdf. If licensed using per Limited Use Socket licensing you can deploy on servers up to 4 sockets. You must acquire a separate user license for each authorized user of this product, with a minimum purchase of five users per server.

For more information, refer to www.ibm.com/software/data/db2/linux-unix-windows/edition-workgroup.html

**DB2 Express-C**

Provides all the core capabilities of DB2 at no charge. Easy to use and embed.

DB2 Express-C is a free, entry-level edition of the DB2 data server for the developer and partner community. It is designed to be up and running in minutes, is easy-to-use and embed, includes self-management features, and embodies all of the core capabilities of DB2 for Linux, UNIX, and Windows such as pureXML. Solutions developed using DB2 Express-C can be seamlessly deployed using more scalable DB2 editions without modifications to the application code.

Program charges: DB2 Express-C can be used for development and deployment at no charge, and can also be distributed with third-party solutions without any royalties to IBM. It can be installed on physical or virtual systems with any amount of CPU and RAM, and is optimized to utilize up to a maximum of two processor cores and 4 GB of memory. DB2 Express-C is refreshed at major release milestones and comes with online community-based assistance. Users requiring more formal support, access to fix packs, or additional capabilities such as high availability clustering and replication features, can purchase optional yearly subscription for DB2 Express® Edition (FTL) or upgrade to other DB2 editions.

For more information, refer to www.ibm.com/software/data/db2/express/.

**DB2 Express Edition**

Fully-functional edition of DB2 at an attractive entry-level price for small and medium businesses.

DB2 Express Edition is a full-function DB2 data server, which provides very attractive entry-level pricing for the small and medium business (SMB) market. It is offered in per Authorized User, Value Unit, or Limited Use Virtual Server based pricing models to provide choices to match SMB customer needs. It comes with simplified packaging and is easy to transparently install within an application. DB2 Express Edition can also be licensed on a yearly fixed term Limited Use Virtual Server license. While it is easy to upgrade to the other editions of DB2 database products, DB2 Express Edition includes the same autonomic manageability features of the more scalable editions. You never have to change your application code to upgrade; simply install the license certificate to upgrade.

Program charges: DB2 Express Edition can be deployed on pervasive SMB operating systems, such as Linux, Windows or Solaris and includes pureXML, web services federation, DB2 Homogeneous Federation, Homogeneous SQL Replication, and backup compression. If licensed as a yearly subscription (DB2 Express Edition FTL) it also includes High Availability feature as long as both primary and secondary servers in the high availability cluster are licensed. If licensed under the Limited Use Virtual Server metric, DB2 Express Edition will use up to four cores on the

server. The DB2 data server cannot use more than 8 GB of memory per server. You must acquire a separate user license for each authorized user of this product with a minimum purchase of five users per server.

For more information, refer to www.ibm.com/software/data/db2/linux-unix-windows/edition-express.html.

**IBM Database Enterprise Developer Edition**

This edition offers a package for a single application developer to design, build, and prototype applications for deployment on any of the IBM Information Management client or server platforms. This comprehensive developer offering includes DB2 Workgroup Server Edition, DB2 Enterprise Server Edition, IDS Enterprise Edition, IBM Database Enterprise Developer Edition, DB2 Connect™ Unlimited Edition for System z®, and all the DB2 Version 10.1 features, allowing you to build solutions that utilize the latest data server technologies.

Program charges: The software in this package cannot be used for production systems. You must acquire a separate user license for each Authorized User of this product.

# Chapter 2. Functionality in DB2 features and DB2 product editions

Some functionality is available in only certain DB2 database product editions. In some cases, the functionality is associated with a particular DB2 feature.

The table indicates which functionality is included in a DB2 product edition. If the functionality is not included but it is available in a DB2 feature, the name of the feature is specified. You must obtain a license for that DB2 feature as well as for the DB2 database product edition.

**Note:** This table is for informational purposes only. For details of entitlement, rights and obligations, refer to the license agreement for your DB2 product.

*Table 1. Functionality in DB2 features and DB2 database product editions*

| Functionality | DB2 Express-C | DB2 Express Edition[1] | DB2 Workgroup Server Edition | DB2 Enterprise Server Edition[2] | DB2 Advanced Enterprise Server Edition[2] | IBM Database Enterprise Developer Edition |
|---|---|---|---|---|---|---|
| Adaptive Compression and classic row compression | No | No | No | DB2 Storage Optimization Feature[4] | Yes | Yes |
| Compression: backup | Yes | Yes | Yes | Yes | Yes | Yes |
| Connection concentrator | No | No | No | Yes | Yes | Yes |
| Continuous Data Ingest | No | No | No | No | Yes | Yes |
| DB2 Advanced Copy Services | No | Yes | Yes | Yes | Yes | Yes |
| functionality | No | No | No | No | No | Yes |
| DB2 Governor | No | No | No | Yes | Yes | Yes |
| DB2 pureScale® functionality | No | No | You are entitled to use DB2 pureScale Feature in a maximum of 16 cores and 64 GB total cluster size. | DB2 pureScale Feature[4] | DB2 pureScale Feature[4] | Yes |
| Federation with DB2 LUW and Informix® Data Server data sources | Yes | Yes | Yes | Yes | Yes | Yes |
| Federation with DB2 LUW and Oracle data sources | No | No | No | No | Yes | Yes |
| High availability disaster recovery | No | Yes | Yes | Yes | Yes | Yes |
| IBM Data Studio | Yes | Yes | Yes | Yes | Yes | Yes |
| IBM InfoSphere® Data Architect | No | No | No | No | Yes[5] | Yes |
| IBM InfoSphere Optim™ Configuration Manager | No | No | No | No | Yes | Yes |

*Table 1. Functionality in DB2 features and DB2 database product editions (continued)*

| Functionality | DB2 Express-C | DB2 Express Edition[1] | DB2 Workgroup Server Edition | DB2 Enterprise Server Edition[2] | DB2 Advanced Enterprise Server Edition[2] | IBM Database Enterprise Developer Edition |
|---|---|---|---|---|---|---|
| IBM InfoSphere Optim Performance Manager Extended Edition[3] | No | No | No | No | Yes | Yes |
| IBM InfoSphere Optim pureQuery Runtime | No | No | No | No | Yes | Yes |
| IBM InfoSphere Optim Query Workload Tuner | No | No | No | No | Yes | Yes |
| Label-based access control (LBAC) | No | Yes | Yes | Yes | Yes | Yes |
| Materialized query tables (MQTs) | No | No | No | Yes | Yes | Yes |
| Multidimensio- nal clustering (MDC) tables | No | No | No | Yes | Yes | Yes |
| Multi-Temperature Storage | No | No | No | Yes | Yes | Yes |
| Net Search Extender | Yes | Yes | Yes | Yes | Yes | Yes |
| Online reorganization | No | Yes | Yes | Yes | Yes | Yes |
| Oracle Compatibility | Yes | Yes | Yes | Yes | Yes | Yes |
| Partitioning - partitioned database environment [3] | No | No | No | No | No | Yes |
| Partitioning - Table partitioning | No | No | No | Yes | Yes | Yes |
| pureXML storage | Yes | Yes | Yes | Yes | Yes | Yes |
| Q Replication with two other DB2 LUW servers | No | No | No | No | Yes | Yes |
| Query parallelism | No | No | No | Yes | Yes | Yes |
| Replication tools | Yes[6] | Yes[6] | Yes[6] | Yes[6] | Yes[6] | Yes |
| Row and column access control (RCAC) | No | Yes | Yes | Yes | Yes | Yes |
| Spatial Extender | Yes | Yes | Yes | Yes | Yes | Yes |
| SQL Replication between DB2 LUW and Informix Data Server | No | Yes | Yes | Yes | Yes | Yes |
| Sybase Compatibility | No | No | No | No | No | No |
| Time Travel Query | Yes | Yes | Yes | Yes | Yes | Yes |
| Tivoli Storage FlashCopy Manager | No | Yes | Yes | Yes | Yes | Yes |
| IBM Tivoli System Automation for Multiplatforms | No | Yes | Yes | Yes | Yes | Yes |
| Workload management | No | No | No | No | Yes | Yes |

**Note:**

1. DB2 Express Edition including DB2 Express Edition Fixed Term License
2. You can purchase all of the DB2 features that are listed in this column for use with IBM InfoSphere Warehouse Enterprise Base and Enterprise Edition products.
3. Partitioned database environment is also bundled with all editions of IBM InfoSphere Warehouse.
4. Separately priced feature.
5. DB2 Advanced Enterprise Server Edition includes 10 InfoSphere Data Architect user licenses.
6. Replication tools except the Replication Center are available on all supported operating systems. The Replication Center is available only on Linux and Windows operating systems.

# Chapter 3. IBM DB2 pureScale Feature

In a competitive, ever-changing global business environment, you cannot afford to let your IT infrastructure slow you down. This reality demands IT systems that provide capacity as needed, exceptional levels of availability, and transparency toward your existing applications.

When workloads grow, does your distributed database system require you to change your applications or change how data is distributed? If so, your system does not scale transparently. Even simple application changes incur time and cost penalties and can pose risks to system availability. The stakes are always high: Every second lost in system availability can have a direct bearing on customer retention, compliance with service level agreements, and your bottom line.

The IBM DB2 pureScale Feature might help reduce the risk and cost associated with growing your distributed database solution by providing extreme capacity and application transparency. Designed for continuous availability-high availability capable of exceeding even the strictest industry standard-this feature tolerates both planned maintenance and component failure with ease.

With the DB2 pureScale Feature, scaling your database solution is simple. Multiple database servers, known as members, process incoming database requests; these members operate in a clustered system and share data. You can transparently add more members to scale out to meet even the most demanding business needs. There are no application changes to make, data to redistribute, or performance tuning to do.

To deliver on a design capable of exceptional levels of database availability, the DB2 pureScale Feature builds on familiar and proven design features from DB2 for z/OS® database software. By also integrating several advanced hardware and software technologies, the DB2 pureScale Feature supports the strictest requirements for high fault tolerance and can sustain processing of database requests even under extreme circumstances.

In the sections that follow, you can learn more about these design features and benefits of the DB2 pureScale Feature:

## Extreme capacity

The IBM DB2 pureScale Feature can scale with near-linear efficiency and high predictability. Adding capacity is as simple as adding new members to the instance.

### High scalability

During testing with typical web commerce and OLTP workloads, the DB2 pureScale Feature demonstrated that it can scale to different levels with exceptional efficiency; the maximum supported configuration provides extreme capacity. To scale out, your existing applications do not have to be aware of the topology of

your DB2 pureScale environment.[1]
When two more members join the instance, they immediately begin processing

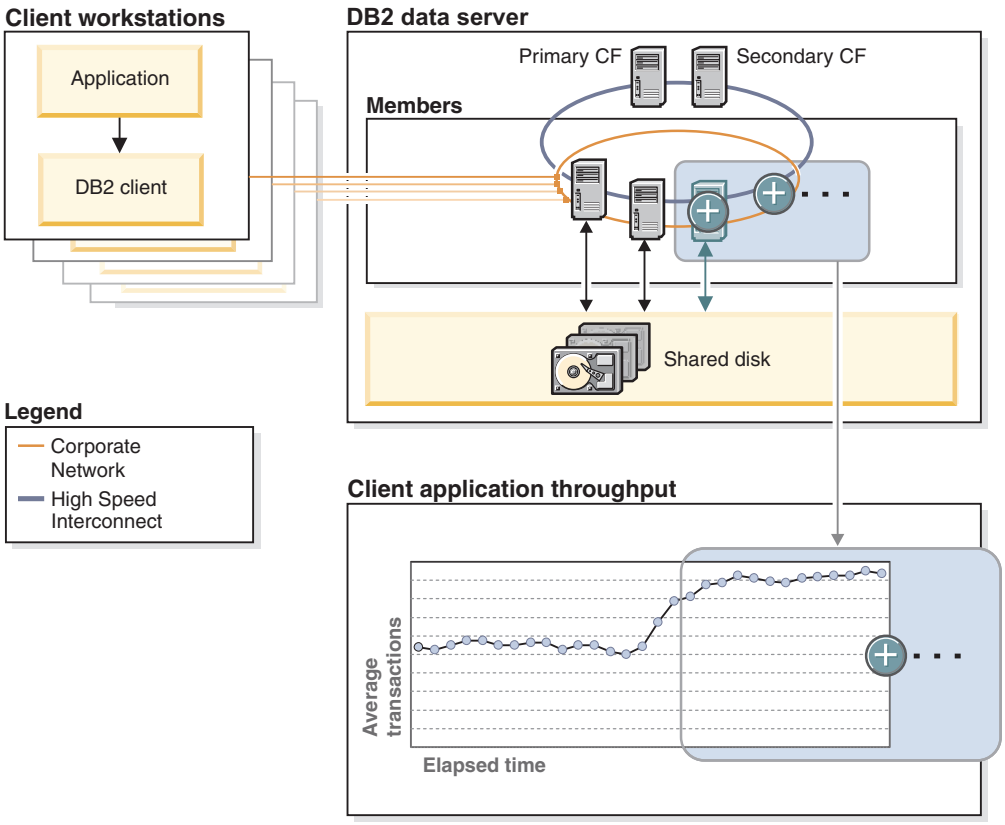



*Figure 1. Scalability of a DB2 pureScale environment. Additional members begin processing incoming database requests as soon as they join the instance. Overall throughput almost doubles as the number of members doubles.*

incoming database requests. Overall throughput almost doubles as the number of members doubles. For more information about scalability, see the DB2 pureScale Feature road map.

## Scalability by design

Why does the DB2 pureScale Feature scale so well? The answer lies in the highly efficient design, which tightly integrates several advanced hardware and software technologies.

For example, the cluster caching facility (CF) handles instance-wide lock management and global caching with great efficiency. Without the equivalent of such a dedicated component to handle locking and caching, the database servers in a cluster must communicate with each other to maintain vital locking and data consistency information. Each time that a database server is added, the amount of communication "chatter" increases, reducing scale-out efficiency.

Even in the maximum supported configuration, your DB2 pureScale environment communicates efficiently. Data pages in the group buffer pool (global cache) are shared between members and the cluster caching facility through Remote Direct

1. During testing, database requests were workload balanced across members by the DB2 pureScale Feature, not routed. Update and select operations were randomized to ensure that the location of data on the shared disk storage had no effect on scalability.

Memory Access (RDMA), without requiring any processor time or I/O cycles on members. All operations are performed over the InfiniBand high-speed interconnect and do not require context switching or routing through a slower IP network stack. Round-trip communication times between cluster components are typically measured in the low tens of microseconds. The end result is an instance that is always aware of what data is in flight and where, but without the performance penalty.

## Continuous availability

Whether it is planned system maintenance or an extreme circumstance, such as when multiple components fail simultaneously, the IBM DB2 pureScale Feature is designed to continue processing incoming database requests without interruption. Automatic load balancing across all active members means optimal resource utilization at all times, which helps to keep application response times low.

### Unplanned events

A sudden software or hardware failure can be highly disruptive, even in a system that employs redundant components. The DB2 pureScale Feature incorporates several design features to deliver fault tolerance that not only can keep your instance available but also minimizes the effect of component failures on the rest of the database system.

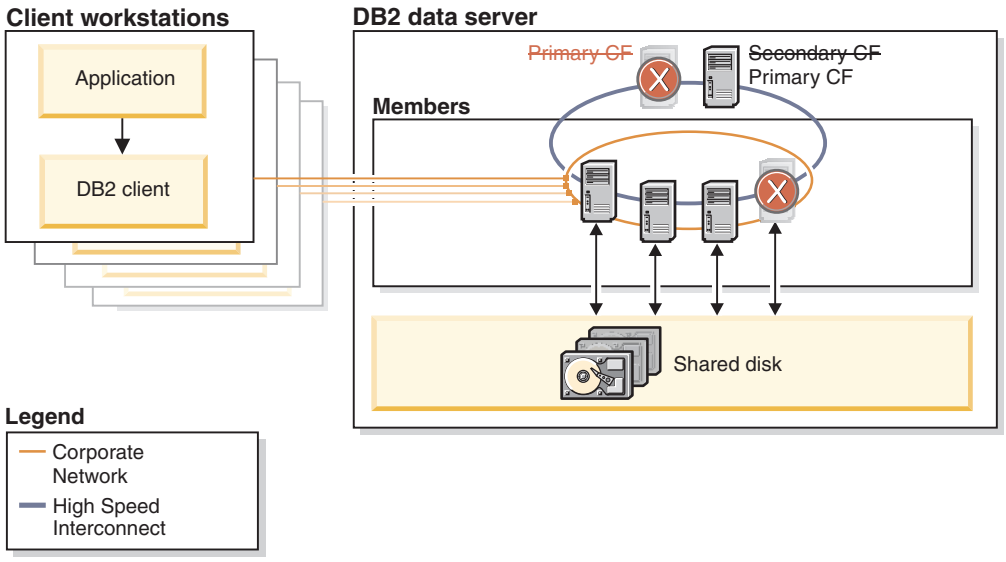



*Figure 2. Component failures in a DB2 pureScale environment; database requests continue to be processed.*

Robust heartbeat detection ensures that failed components are identified and isolated rapidly. Recovery from component failures is fully automatic and requires no intervention.

If a member fails while processing database requests, it is immediately fenced off from the rest of the system. During the failure, most of your data on the shared disk storage remains available to active members processing database requests. Only the data that was inflight on the failed member is temporarily held by a retained lock until the DB2 pureScale Feature completes the automated member crash recovery.

After a software failure, the member is restarted on its home host, and recovery is performed. The member resumes transaction processing as soon as recovery is complete. After a hardware failure, the member restarts on another host (a process known as *restart light*) so that the data can be recovered. As soon as its home host is available again, the member fails back to that host, restarts, and resumes processing.

After a software or hardware failure on the primary cluster caching facility, a secondary, duplexed cluster caching facility automatically takes over the primary role. This takeover is transparent to applications and causes only minimal delay because of the continuous duplexing of locking and caching information between cluster caching facilities. The instance remains available.

### Planned events

System maintenance in a DB2 pureScale environment is designed to cause as little disruption as possible. You can roll out system upgrades without stopping the DB2 pureScale instance or affecting database availability.

To perform system maintenance on a member, you quiesce it. After existing transactions on the member are completed (drained), you take the member offline and perform the system maintenance. During the maintenance period, new transaction requests are automatically directed to other, active members, a process that is transparent to applications.

After the maintenance is complete and you restart the member, it begins processing database transactions again as soon as it rejoins the instance.

## Application transparency

Getting started with the IBM DB2 pureScale Feature is quick and simple: Applications do not have to be aware of the topology of your database environment when you deploy the feature. This means that applications work just as they did before, yet they can benefit

from the extreme capacity and continuous availability from the moment that you start your DB2 pureScale instance for the first time.

### Increasing capacity

Capacity planning with the DB2 pureScale Feature is simple. You can start small and add members to your database environment as your needs grow, scaling out from the most basic highly available configuration all the way to the maximum supported configuration, which provides extreme processing capacity. Scaling is near linear in efficiency and highly predictable.

When you scale out, no application changes or repartitioning of your data is required. No performance tuning is required to scale efficiently. If you need more capacity, you simply add more members.

### Maintaining availability

Maintaining database availability means both compliance with service level agreements (SLAs) and high tolerance to component failures. To maximize hardware utilization rates and help keep response times consistent for your applications, incoming database requests are automatically load balanced across all

active members in your DB2 pureScale instance. To minimize the impact of component failures, the automated restart and recovery process of the DB2 pureScale Feature runs quickly and without affecting most database requests. Only those database requests that were being processed by a failed member must be resubmitted by the originating application; the resubmitted requests are then processed by the next available member.

In the example in the following diagram, several events take place in short succession. Multiple component failures require automated, internal recovery, and a scale-out operation increases the capacity of the DB2 pureScale instance. None of these events requires any application awareness. The box containing the components of the DB2 pureScale Feature is shaded to indicate application transparency.

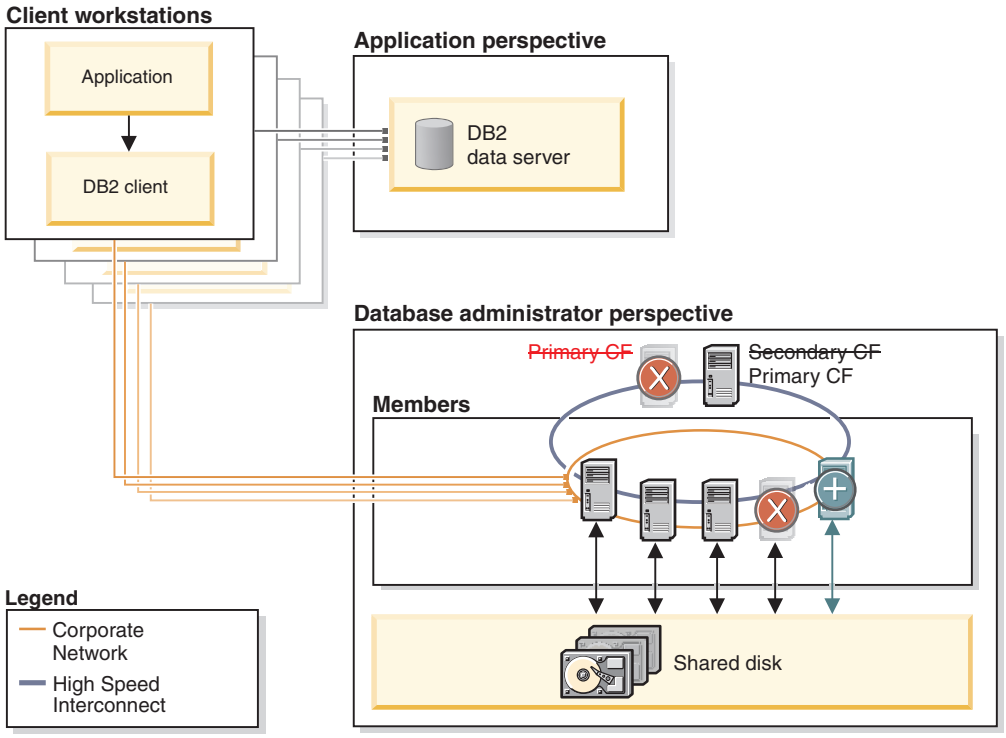



*Figure 3. A DB2 pureScale environment encountering multiple component failures and being scaled out. Applications connecting to the database need not be aware of these events.*

## Planning made simple

The ability to easily add and remove resources helps you to manage challenges such as the following ones:

- Cyclical workloads. If some of your workloads are cyclical (for example, seasonal), you can add resources before they are required, and then move the extra capacity somewhere else later on.
- Sudden increases in workloads. An SLA might dictate minimum response times for completing database requests. If you discover sudden workload surges from some applications that are threatening response times, you can help meet your SLA by quickly moving additional members to the database that is experiencing the peak demand.
- Maintenance-related slowdowns. To help negate the effect of system maintenance on the overall throughput of your DB2 pureScale environment, you

can add amember to your environment before commencing maintenance on an existing member. After you complete the system maintenance and the original member rejoins the instance, you can remove the additional resource or perform maintenance on other members.

## How does a DB2 pureScale environment compare to a multi-partition database environment?

The IBM DB2 pureScale Feature, much like a multi-partition database environment, provides a scalable and highly available database solution. However, the instance type and data layout of a DB2 pureScale environment and a multi-partition database environment are different.

Each environment has unique advantages and is tailored for specific types of database processing. Which environment you use depends on your specific business requirements. The following table outlines some of the key differences between a multi-partition database environment and a DB2 pureScale environment.

*Table 2. Comparison of a multi-partition database environment and a DB2 pureScale environment*

| | **Multi-partition database environment** | **DB2 pureScale environment** |
|---|---|---|
| Provides | A multi-partition database environment provides members working in parallel in a shared-nothing environment. Data is distributed across partitions, with local access to a subset of data by each member. Typically, a query is processed by multiple members simultaneously. | A DB2 pureScale environment provides DB2 members that independently process database requests but access data stored on a shared disk. Each SQL request is executed by a single member. |
| Designed for | • Faster response time of long-running or complex queries<br>• Management of very large data sets<br>• I/O spread across many computers | • A continuously available database solution<br>• Increased throughput of many concurrent short queries<br>• Easy scalability |
| Transaction size and type | The ideal scenario for a multi-partition database environment is lengthy workloads that can be subdivided and directed to specific partitions to run in parallel.<br><br>A multi-partition database environment is ideal for applications that are part of a decision support system (DSS), for business intelligence applications, and for data warehousing. | A DB2 pureScale environment is ideal for short transactions where there is little need to parallelize each query. Queries are automatically routed to different members, based on member workload.<br><br>The ideal scenario for a DB2 pureScale environment includes workloads that handle online transaction processing (OLTP) or enterprise resource planning (ERP). |

# Chapter 4. IBM DB2 Storage Optimization Feature

This feature includes data row compression and other compression types to help maximize the use of existing storage.

*Data row compression* uses a table-level compression dictionary to compress data by row. Compressing data at the row level is advantageous because it allows repeating patterns that span multiple column values within a row to be replaced with shorter symbol strings, allowing for improved performance. In I/O-bound systems, it will not only reduce storage requirements but may improve overall performance.

*Index compression* utilizes multiple algorithms to reduce the storage requirements for indexes and reduces the total storage footprint of your DB2 database.

*Temp compression* will automatically compress the amount of space used for temp tables. This automated compression will help to both improve performance of business intelligence workloads using large temp tables as well as reducing the total storage needed by DB2 database systems.

This feature is available as an option for DB2 Enterprise Server Edition only and can only be acquired if the underlying DB2 data server is licensed with the Processor Value Unit charge metric.

For more information, refer to www.ibm.com/software/data/db2/linux-unix-windows/editions_features_storage.html.

# Part 2. Security

DB2 database products provide security features that you can use to protect your sensitive data. With the number of both internal and external security threats growing, it is important to separate the tasks of keeping data secure from the management tasks of administering critical systems.

# Chapter 5. DB2 security model

Two modes of security control access to the DB2 database system data and functions. Access to the DB2 database system is managed by facilities that reside outside the DB2 database system (authentication), whereas access within the DB2 database system is managed by the database manager (authorization).

## Authentication

Authentication is the process by which a system verifies a user's identity. User authentication is completed by a security facility outside the DB2 database system, through an authentication security plug-in module. A default authentication security plug-in module that relies on operating-system-based authentication is included when you install the DB2 database system. For your convenience, the DB2 database manager also ships with authentication plug-in modules for Kerberos and lightweight directory access protocol (LDAP). To provide even greater flexibility in accommodating your specific authentication needs, you can build your own authentication security plug-in module.

The authentication process produces a DB2 authorization ID. Group membership information for the user is also acquired during authentication. Default acquisition of group information relies on an operating-system based group-membership plug-in module that is included when you install the DB2 database system. If you prefer, you can acquire group membership information by using a specific group-membership plug-in module, such as LDAP.

## Authorization

After a user is authenticated, the database manager determines if that user is allowed to access DB2 data or resources. Authorization is the process whereby the DB2 database manager obtains information about the authenticated user, indicating which database operations that user can perform, and which data objects that user can access.

The different sources of permissions available to an authorization ID are as follows:

1. Primary permissions: those granted to the authorization ID directly.
2. Secondary permissions: those granted to the groups and roles in which the authorization ID is a member.
3. Public permissions: those granted to PUBLIC.
4. Context-sensitive permissions: those granted to a trusted context role.

Authorization can be given to users in the following categories:

- System-level authorization

  The system administrator (SYSADM), system control (SYSCTRL), system maintenance (SYSMAINT), and system monitor (SYSMON) authorities provide varying degrees of control over instance-level functions. Authorities provide a way both to group privileges and to control maintenance and utility operations for instances, databases, and database objects.
- Database-level authorization

  The security administrator (SECADM), database administrator (DBADM), access control (ACCESSCTRL), data access (DATAACCESS), SQL administrator

(SQLADM), workload management administrator (WLMADM), and explain (EXPLAIN) authorities provide control within the database. Other database authorities include LOAD (ability to load data into a table), and CONNECT (ability to connect to a database).

- Object-level authorization

  Object level authorization involves checking privileges when an operation is performed on an object. For example, to select from a table a user must have SELECT privilege on a table (as a minimum).

- Content-based authorization

  Views provide a way to control which columns or rows of a table specific users can read. Label-based access control (LBAC) determines which users have read and write access to individual rows and individual columns.

You can use these features, in conjunction with the DB2 audit facility for monitoring access, to define and manage the level of security your database installation requires.

# Chapter 6. Authentication methods for your server

Access to an instance or a database first requires that the user be *authenticated*. The *authentication type* for each instance determines how and where a user will be verified.

The authentication type is stored in the configuration file at the server. It is initially set when the instance is created. There is one authentication type per instance, which covers access to that database server and all the databases under its control.

If you intend to access data sources from a federated database, you must consider data source authentication processing and definitions for federated authentication types.

**Note:** You can check the following website for certification information about the cryptographic routines used by the DB2 database management system to perform encryption of the user ID and password when using SERVER_ENCRYPT authentication, and of the user ID, password, and user data when using DATA_ENCRYPT authentication: http://www.ibm.com/security/standards/st_evaluations.shtml.

## Switching User on an Explicit Trusted Connection

For CLI/ODBC and XA CLI/ODBC applications, the authentication mechanism used when processing a switch user request that requires authentication is the same as the mechanism used to originally establish the trusted connection itself. Therefore, any other negotiated security attributes (for example, encryption algorithm, encryption keys, and plug-in names) used during the establishment of the explicit trusted connection are assumed to be the same for any authentication required for a switch user request on that trusted connection. Java™ applications allow the authentication method to be changed on a switch user request (by use of a datasource property).

Because a trusted context object can be defined such that switching user on a trusted connection does *not* require authentication, in order to take full advantage of the switch user on an explicit trusted connection feature, user-written security plug-ins must be able to:

- Accept a user ID-only token
- Return a valid DB2 authorization ID for that user ID

**Note:** An explicit trusted connection cannot be established if the CLIENT type of authentication is in effect.

## Authentication types provided

The following authentication types are provided:

**SERVER**
Specifies that authentication occurs on the server through the security mechanism in effect for that configuration, for example, through a security plug-in module. The default security mechanism is that if a user ID and password are specified during the connection or attachment attempt, they

are sent to the server and compared to the valid user ID and password combinations at the server to determine if the user is permitted to access the instance.

**Note:** The server code detects whether a connection is local or remote. For local connections, when authentication is SERVER, a user ID and password are not required for authentication to be successful.

**SERVER_ENCRYPT**

Specifies that the server accepts encrypted SERVER authentication schemes. If the client authentication is not specified, the client is authenticated using the method selected at the server. The user ID and password are encrypted when they are sent over the network from the client to the server.

When the resulting authentication method negotiated between the client and server is SERVER_ENCRYPT, you can choose to encrypt the user ID and password using an AES (Advanced Encryption Standard) 256-bit algorithm. To do this, set the **alternate_auth_enc** database manager configuration parameter. This configuration parameter has three settings:

- NOT_SPECIFIED (default) means that the server accepts the encryption algorithm that the client proposes, including an AES 256-bit algorithm.
- AES_CMP means that if the connecting client proposes DES but supports AES encryption, the server renegotiates for AES encryption.
- AES_ONLY means that the server accepts only AES encryption. If the client does not support AES encryption, the connection is rejected.

AES encryption can be used only when the authentication method negotiated between the client and server is SERVER_ENCRYPT.

**CLIENT**

Specifies that authentication occurs on the database partition where the application is invoked using operating system security. The user ID and password specified during a connection or attachment attempt are compared with the valid user ID and password combinations on the client node to determine whether the user ID is permitted access to the instance. No further authentication will take place on the database server. This is sometimes called single signon.

If the user performs a local or client login, the user is known only to that local client workstation.

If the remote instance has CLIENT authentication, two other parameters determine the final authentication type: **trust_allclnts** and **trust_clntauth**.

**CLIENT level security for TRUSTED clients only:**

Trusted clients are clients that have a reliable, local security system.

When the authentication type of CLIENT has been selected, an additional option might be selected to protect against clients whose operating environment has no inherent security.

To protect against unsecured clients, the administrator can select Trusted Client Authentication by setting the **trust_allclnts** parameter to NO. This implies that all trusted platforms can authenticate the user on behalf of the server. Untrusted clients are authenticated on the Server and must provide a user ID and

password. You use the **trust_allclnts** configuration parameter to indicate whether you are trusting clients. The default for this parameter is YES.

**Note:** It is possible to trust all clients (**trust_allclnts** is YES) yet have some of those clients as those who do not have a native safe security system for authentication.

You might also want to complete authentication at the server even for trusted clients. To indicate where to validate trusted clients, you use the **trust_clntauth** configuration parameter. The default for this parameter is CLIENT.

**Note:** For trusted clients only, if no user ID or password is explicitly provided when attempting to CONNECT or **ATTACH**, then validation of the user takes place at the client. The **trust_clntauth** parameter is only used to determine where to validate the information provided on the USER or USING clauses.

To protect against all clients, including JCC type 4 clients on z/OS and System i® but excluding native DB2 clients on z/OS, OS/390®, VM, VSE, and System i, set the **trust_allclnts** parameter to DRDAONLY. Only these clients can be trusted to perform client-side authentication. All other clients must provide a user ID and password to be authenticated by the server.

The **trust_clntauth** parameter is used to determine where the clients mentioned previously are authenticated: if **trust_clntauth** is CLIENT, authentication takes place at the client. If **trust_clntauth** is SERVER, authentication takes place at the client when no user ID and password are provided and at the server when a user ID and password are provided.

*Table 3. Authentication Modes using TRUST_ALLCLNTS and TRUST_CLNTAUTH Parameter Combinations.*

| trust_ allclnts | trust_ clntauth | Untrusted non-DRDA® Client Authen-tication (no user ID & password) | Untrusted non-DRDA Client Authen-tication (with user ID & password) | Trusted non-DRDA Client Authen-tication (no user ID & password) | Trusted non-DRDA Client Authen-tication (with user ID & password) | DRDA Client Authen-tication (no user ID & password) | DRDA Client Authen-tication (with user ID & password) |
|---|---|---|---|---|---|---|---|
| YES | CLIENT | CLIENT | CLIENT | CLIENT | CLIENT | CLIENT | CLIENT |
| YES | SERVER | CLIENT | SERVER | CLIENT | SERVER | CLIENT | SERVER |
| NO | CLIENT | SERVER | SERVER | CLIENT | CLIENT | CLIENT | CLIENT |
| NO | SERVER | SERVER | SERVER | CLIENT | SERVER | CLIENT | SERVER |
| DRDAONLY | CLIENT | SERVER | SERVER | SERVER | SERVER | CLIENT | CLIENT |
| DRDAONLY | SERVER | SERVER | SERVER | SERVER | SERVER | CLIENT | SERVER |

**DATA_ENCRYPT**

The server accepts encrypted SERVER authentication schemes and the encryption of user data. The authentication works the same way as that shown with SERVER_ENCRYPT. The user ID and password are encrypted when they are sent over the network from the client to the server.

The following user data are encrypted when using this authentication type:

- SQL and XQuery statements.
- SQL program variable data.
- Output data from the server processing of an SQL or XQuery statement and including a description of the data.
- Some or all of the answer set data resulting from a query.
- Large object (LOB) data streaming.
- SQLDA descriptors.

**DATA_ENCRYPT_CMP**

The server accepts encrypted SERVER authentication schemes and the encryption of user data. In addition, this authentication type allows compatibility with down level products not supporting DATA_ENCRYPT authentication type. These products are permitted to connect with the SERVER_ENCRYPT authentication type and without encrypting user data. Products supporting the new authentication type must use it. This authentication type is only valid in the server's database manager configuration file and is not valid when used on the **CATALOG DATABASE** command.

**KERBEROS**

Used when both the DB2 client and server are on operating systems that support the Kerberos security protocol. The Kerberos security protocol performs authentication as a third party authentication service by using conventional cryptography to create a shared secret key. This key becomes a user's credential and is used to verify the identity of users during all occasions when local or network services are requested. The key eliminates the need to pass the user name and password across the network as clear text. Using the Kerberos security protocol enables the use of a single sign-on to a remote DB2 database server. The KERBEROS authentication type is supported on various operating systems.

Kerberos authentication works as follows:

1. A user logging on to the client machine using a domain account authenticates to the Kerberos key distribution center (KDC) at the domain controller. The key distribution center issues a ticket-granting ticket (TGT) to the client.
2. During the first phase of the connection the server sends the target principal name, which is the service account name for the DB2 database server service, to the client. Using the server's target principal name and the target-granting ticket, the client requests a service ticket from the ticket-granting service (TGS) which also resides at the domain controller. If both the client's ticket-granting ticket and the server's target principal name are valid, the TGS issues a service ticket to the client. The principal name recorded in the database directory can be specified as name/instance@REALM. (This is in addition to DOMAIN\userID and userID@xxx.xxx.xxx.com formats accepted on Windows.)
3. The client sends this service ticket to the server using the communication channel (which can be, as an example, TCP/IP).
4. The server validates the client's server ticket. If the client's service ticket is valid, then the authentication is completed.

It is possible to catalog the databases on the client machine and explicitly specify the Kerberos authentication type with the server's target principal name. In this way, the first phase of the connection can be bypassed.

If a user ID and a password are specified, the client will request the ticket-granting ticket for that user account and use it for authentication.

**KRB_SERVER_ENCRYPT**
Specifies that the server accepts KERBEROS authentication or encrypted SERVER authentication schemes. If the client authentication is KERBEROS, the client is authenticated using the Kerberos security system. If the client authentication is SERVER_ENCRYPT, the client is authenticated using a user ID and encryption password. If the client authentication is not specified, then the client will use Kerberos if available, otherwise it will use password encryption. For other client authentication types, an authentication error is returned. The authentication type of the client cannot be specified as KRB_SERVER_ENCRYPT

**Note:** The Kerberos authentication types are supported on clients and servers running on specific operating systems. For Windows operating systems, both client and server machines must either belong to the same Windows domain or belong to trusted domains. This authentication type should be used when the server supports Kerberos and some, but not all, of the client machines support Kerberos authentication.

**GSSPLUGIN**
Specifies that the server uses a GSS-API plug-in to perform authentication. If the client authentication is not specified, the server returns a list of server-supported plug-ins, including any Kerberos plug-in that is listed in the **srvcon_gssplugin_list** database manager configuration parameter, to the client. The client selects the first plug-in found in the client plug-in directory from the list. If the client does not support any plug-in in the list, the client is authenticated using the Kerberos authentication scheme (if it is returned). If the client authentication is the GSSPLUGIN authentication scheme, the client is authenticated using the first supported plug-in in the list.

**GSS_SERVER_ENCRYPT**
Specifies that the server accepts plug-in authentication or encrypted server authentication schemes. If client authentication occurs through a plug-in, the client is authenticated using the first client-supported plug-in in the list of server-supported plug-ins.

If the client authentication is not specified and an implicit connect is being performed (that is, the client does not supply a user ID and password when making the connection), the server returns a list of server-supported plug-ins, the Kerberos authentication scheme (if one of the plug-ins in the list is Kerberos-based), and the encrypted server authentication scheme. The client is authenticated using the first supported plug-in found in the client plug-in directory. If the client does not support any of the plug-ins that are in the list, the client is authenticated using the Kerberos authentication scheme. If the client does not support the Kerberos authentication scheme, the client is authenticated using the encrypted server authentication scheme, and the connection will fail because of a missing password. A client supports the Kerberos authentication scheme if a DB2 supplied Kerberos plug-in exists for the operating system, or a Kerberos-based plug-in is specified for the **srvcon_gssplugin_list** database manager configuration parameter.

If the client authentication is not specified and an explicit connection is being performed (that is, both the user ID and password are supplied), the authentication type is equivalent to SERVER_ENCRYPT. In this case, the choice of the encryption algorithm used to encrypt the user ID and password depends on the setting of the **alternate_auth_enc** database manager configuration parameter.

**Note:**

1. Do not inadvertently lock yourself out of your instance when you are changing the authentication information, since access to the configuration file itself is protected by information in the configuration file. The following database manager configuration file parameters control access to the instance:
   - **authentication** *
   - **sysadm_group** *
   - **trust_allclnts**
   - **trust_clntauth**
   - **sysctrl_group**
   - **sysmaint_group**

   * Indicates the two most important parameters.

   There are some things that can be done to ensure this does not happen: If you do accidentally lock yourself out of the DB2 database system, you have a fail-safe option available on all platforms that will allow you to override the usual DB2 database security checks to update the database manager configuration file using a highly privileged local operating system security user. This user *always* has the privilege to update the database manager configuration file and thereby correct the problem. However, this security bypass is restricted to a local update of the database manager configuration file. You cannot use a fail-safe user remotely or for any other DB2 database command. This special user is identified as follows:
   - UNIX platforms: the instance owner
   - Windows platform: someone belonging to the local "Administrators" group
   - Other platforms: there is no local security on the other platforms, so all users pass local security checks anyway

# Chapter 7. Authorization, privileges, and object ownership

Users (identified by an authorization ID) can successfully execute operations only if they have the authority to perform the specified function. To create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so forth.

The database manager requires that each user be specifically authorized to use each database function needed to perform a specific task. A user can acquire the necessary authorization through a grant of that authorization to their user ID or through membership in a role or a group that holds that authorization.

There are three forms of authorization, *administrative authority*, *privileges*, and *LBAC credentials*. In addition, ownership of objects brings with it a degree of authorization on the objects created. These forms of authorization are discussed in the following section.

## Administrative authority

The person or persons holding administrative authority are charged with the task of controlling the database manager and are responsible for the safety and integrity of the data.

**System-level authorization**

The system-level authorities provide varying degrees of control over instance-level functions:

- SYSADM (system administrator) authority

  The SYSADM (system administrator) authority provides control over all the resources created and maintained by the database manager. The system administrator possesses all the authorities of SYSCTRL, SYSMAINT, and SYSMON authority. The user who has SYSADM authority is responsible both for controlling the database manager, and for ensuring the safety and integrity of the data.
- SYSCTRL authority

  The SYSCTRL authority provides control over operations that affect system resources. For example, a user with SYSCTRL authority can create, update, start, stop, or drop a database. This user can also start or stop an instance, but cannot access table data. Users with SYSCTRL authority also have SYSMON authority.
- SYSMAINT authority

  The SYSMAINT authority provides the authority required to perform maintenance operations on all databases associated with an instance. A user with SYSMAINT authority can update the database configuration, backup a database or table space, restore an existing database, and monitor a database. Like SYSCTRL, SYSMAINT does not provide access to table data. Users with SYSMAINT authority also have SYSMON authority.
- SYSMON (system monitor) authority

  The SYSMON (system monitor) authority provides the authority required to use the database system monitor.

**Database-level authorization**

The database level authorities provide control within the database:

- DBADM (database administrator)

  The DBADM authority level provides administrative authority over a single database. This database administrator possesses the privileges required to create objects and issue database commands.

  The DBADM authority can be granted only by a user with SECADM authority. The DBADM authority cannot be granted to PUBLIC.

- SECADM (security administrator)

  The SECADM authority level provides administrative authority for security over a single database. The security administrator authority possesses the ability to manage database security objects (database roles, audit policies, trusted contexts, security label components, and security labels) and grant and revoke all database privileges and authorities. A user with SECADM authority can transfer the ownership of objects that they do not own. They can also use the AUDIT statement to associate an audit policy with a particular database or database object at the server.

  The SECADM authority has no inherent privilege to access data stored in tables. It can only be granted by a user with SECADM authority. The SECADM authority cannot be granted to PUBLIC.

- SQLADM (SQL administrator)

  The SQLADM authority level provides administrative authority to monitor and tune SQL statements within a single database. It can be granted by a user with ACCESSCTRL or SECADM authority.

- WLMADM (workload management administrator)

  The WLMADM authority provides administrative authority to manage workload management objects, such as service classes, work action sets, work class sets, and workloads. It can be granted by a user with ACCESSCTRL or SECADM authority.

- EXPLAIN (explain authority)

  The EXPLAIN authority level provides administrative authority to explain query plans without gaining access to data. It can only be granted by a user with ACCESSCTRL or SECADM authority.

- ACCESSCTRL (access control authority)

  The ACCESSCTRL authority level provides administrative authority to issue the following GRANT (and REVOKE) statements.

  – GRANT (Database Authorities)

    ACCESSCTRL authority does not give the holder the ability to grant ACCESSCTRL, DATAACCESS, DBADM, or SECADM authority. Only a user who has SECADM authority can grant these authorities.

  – GRANT (Global Variable Privileges)
  – GRANT (Index Privileges)
  – GRANT (Module Privileges)
  – GRANT (Package Privileges)
  – GRANT (Routine Privileges)
  – GRANT (Schema Privileges)
  – GRANT (Sequence Privileges)
  – GRANT (Server Privileges)
  – GRANT (Table, View, or Nickname Privileges)
  – GRANT (Table Space Privileges)

- GRANT (Workload Privileges)
- GRANT (XSR Object Privileges)

ACCESSCTRL authority can only be granted by a user with SECADM authority. The ACCESSCTRL authority cannot be granted to PUBLIC.

- DATAACCESS (data access authority)

  The DATAACCESS authority level provides the following privileges and authorities.
  - LOAD authority
  - SELECT, INSERT, UPDATE, DELETE privilege on tables, views, nicknames, and materialized query tables
  - EXECUTE privilege on packages
  - EXECUTE privilege on modules
  - EXECUTE privilege on routines

    Except on the audit routines: AUDIT_ARCHIVE, AUDIT_LIST_LOGS, AUDIT_DELIM_EXTRACT.
  - READ privilege on all global variables and WRITE privilege on all global variables except variables which are read-only
  - USAGE privilege on all XSR objects
  - USAGE privilege on all sequences

  It can be granted only by a user who holds SECADM authority. The DATAACCESS authority cannot be granted to PUBLIC.
- Database authorities (non-administrative)

  To perform activities such as creating a table or a routine, or for loading data into a table, specific database authorities are required. For example, the LOAD database authority is required for use of the **load** utility to load data into tables (a user must also have INSERT privilege on the table).

## Privileges

A privilege is a permission to perform an action or a task. Authorized users can create objects, have access to objects they own, and can pass on privileges on their own objects to other users by using the GRANT statement.

Privileges may be granted to individual users, to groups, or to PUBLIC. PUBLIC is a special group that consists of all users, including future users. Users that are members of a group will indirectly take advantage of the privileges granted to the group, where groups are supported.

*The CONTROL privilege*: Possessing the CONTROL privilege on an object allows a user to access that database object, and to grant and revoke privileges to or from other users on that object.

**Note:** The CONTROL privilege only apples to tables, views, nicknames, indexes, and packages.

If a different user requires the CONTROL privilege to that object, a user with SECADM or ACCESSCTRL authority could grant the CONTROL privilege to that object. The CONTROL privilege cannot be revoked from the object owner, however, the object owner can be changed by using the TRANSFER OWNERSHIP statement.

*Individual privileges*: Individual privileges can be granted to allow a user to carry out specific tasks on specific objects. Users with the administrative authorities ACCESSCTRL or SECADM, or with the CONTROL privilege, can grant and revoke privileges to and from users.

Individual privileges and database authorities allow a specific function, but do not include the right to grant the same privileges or authorities to other users. The right to grant table, view, schema, package, routine, and sequence privileges to others can be extended to other users through the WITH GRANT OPTION on the GRANT statement. However, the WITH GRANT OPTION does not allow the person granting the privilege to revoke the privilege once granted. You must have SECADM authority, ACCESSCTRL authority, or the CONTROL privilege to revoke the privilege.

*Privileges on objects in a package or routine*: When a user has the privilege to execute a package or routine, they do not necessarily require specific privileges on the objects used in the package or routine. If the package or routine contains static SQL or XQuery statements, the privileges of the owner of the package are used for those statements. If the package or routine contains dynamic SQL or XQuery statements, the authorization ID used for privilege checking depends on the setting of the **DYNAMICRULES BIND** option of the package issuing the dynamic query statements, and whether those statements are issued when the package is being used in the context of a routine (except on the audit routines: AUDIT_ARCHIVE, AUDIT_LIST_LOGS, AUDIT_DELIM_EXTRACT).

A user or group can be authorized for any combination of individual privileges or authorities. When a privilege is associated with an object, that object must exist. For example, a user cannot be given the SELECT privilege on a table unless that table has previously been created.

**Note:** Care must be taken when an authorization name representing a user or a group is granted authorities and privileges and there is no user, or group created with that name. At some later time, a user or a group can be created with that name and automatically receive all of the authorities and privileges associated with that authorization name.

The REVOKE statement is used to revoke previously granted privileges. The revoking of a privilege from an authorization name revokes the privilege granted by all authorization names.

Revoking a privilege from an authorization name does not revoke that same privilege from any other authorization names that were granted the privilege by that authorization name. For example, assume that CLAIRE grants SELECT WITH GRANT OPTION to RICK, then RICK grants SELECT to BOBBY and CHRIS. If CLAIRE revokes the SELECT privilege from RICK, BOBBY and CHRIS still retain the SELECT privilege.

## LBAC credentials

Label-based access control (LBAC) lets the security administrator decide exactly who has write access and who has read access to individual rows and individual columns. The security administrator configures the LBAC system by creating security policies. A security policy describes the criteria used to decide who has access to what data. Only one security policy can be used to protect any one table but different tables can be protected by different security policies.

After creating a security policy, the security administrator creates database objects, called security labels and exemptions that are part of that policy. A security label describes a certain set of security criteria. An exemption allows a rule for comparing security labels not to be enforced for the user who holds the exemption, when they access data protected by that security policy.

Once created, a security label can be associated with individual columns and rows in a table to protect the data held there. Data that is protected by a security label is called protected data. A security administrator allows users access to protected data by granting them security labels. When a user tries to access protected data, that user's security label is compared to the security label protecting the data. The protecting label blocks some security labels and does not block others.

## Object ownership

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership means the user is authorized to reference the object in any applicable SQL or XQuery statement.

When an object is created within a schema, the authorization ID of the statement must have the required privilege to create objects in the implicitly or explicitly specified schema. That is, the authorization name must either be the owner of the schema, or possess the CREATEIN privilege on the schema.

**Note:** This requirement is not applicable when creating table spaces, buffer pools or database partition groups. These objects are not created in schemas.

When an object is created, the authorization ID of the statement is the definer of that object and by default becomes the owner of the object after it is created.

**Note:** One exception exists. If the AUTHORIZATION option is specified for the CREATE SCHEMA statement, any other object that is created as part of the CREATE SCHEMA operation is owned by the authorization ID specified by the AUTHORIZATION option. Any objects that are created in the schema after the initial CREATE SCHEMA operation, however, are owned by the authorization ID associated with the specific CREATE statement.

For example, the statement `CREATE SCHEMA SCOTTSTUFF AUTHORIZATION SCOTT CREATE TABLE T1 (C1 INT)` creates the schema `SCOTTSTUFF` and the table `SCOTTSTUFF.T1`, which are both owned by SCOTT. Assume that the user BOBBY is granted the CREATEIN privilege on the `SCOTTSTUFF` schema and creates an index on the `SCOTTSTUFF.T1` table. Because the index is created after the schema, BOBBY owns the index on `SCOTTSTUFF.T1`.

Privileges are assigned to the object owner based on the type of object being created:

- The CONTROL privilege is implicitly granted on newly created tables, indexes, and packages. This privilege allows the object creator to access the database object, and to grant and revoke privileges to or from other users on that object. If a different user requires the CONTROL privilege to that object, a user with ACCESSCTRL or SECADM authority must grant the CONTROL privilege to that object. The CONTROL privilege cannot be revoked by the object owner.
- The CONTROL privilege is implicitly granted on newly created views if the object owner has the CONTROL privilege on all the tables, views, and nicknames referenced by the view definition.

- Other objects like triggers, routines, sequences, table spaces, and buffer pools do not have a CONTROL privilege associated with them. The object owner does, however, automatically receive each of the privileges associated with the object and those privileges are with the WITH GRANT OPTION, where supported. Therefore the object owner can provide these privileges to other users by using the GRANT statement. For example, if USER1 creates a table space, USER1 automatically has the USEAUTH privilege with the WITH GRANT OPTION on this table space and can grant the USEAUTH privilege to other users. In addition, the object owner can alter, add a comment on, or drop the object. These authorizations are implicit for the object owner and cannot be revoked.

Certain privileges on the object, such as altering a table, can be granted by the owner, and can be revoked from the owner by a user who has ACCESSCTRL or SECADM authority. Certain privileges on the object, such as commenting on a table, cannot be granted by the owner and cannot be revoked from the owner. Use the TRANSFER OWNERSHIP statement to move these privileges to another user. When an object is created, the authorization ID of the statement is the definer of that object and by default becomes the owner of the object after it is created. However, when you use the **BIND** command to create a package and you specify the **OWNER** *authorization id* option, the owner of objects created by the static SQL statements in the package is the value of *authorization id*. In addition, if the AUTHORIZATION clause is specified on a CREATE SCHEMA statement, the authorization name specified after the AUTHORIZATION keyword is the owner of the schema.

A security administrator or the object owner can use the TRANSFER OWNERSHIP statement to change the ownership of a database object. An administrator can therefore create an object on behalf of an authorization ID, by creating the object using the authorization ID as the qualifier, and then using the TRANSFER OWNERSHIP statement to transfer the ownership that the administrator has on the object to the authorization ID.

# Chapter 8. Default privileges granted on creating a database

When you create a database, default database level authorities and default object level privileges are granted to you within that database.

The authorities and privileges that you are granted are listed according to the system catalog views where they are recorded:

1. SYSCAT.DBAUTH
   - The database creator is granted the following authorities:
     - ACCESSCTRL
     - DATAACCESS
     - DBADM
     - SECADM
   - In a non-restrictive database, the special group PUBLIC is granted the following authorities:
     - CREATETAB
     - BINDADD
     - CONNECT
     - IMPLICIT_SCHEMA
2. SYSCAT.TABAUTH

   In a non-restrictive database, the special group PUBLIC is granted the following privileges:
   - SELECT on all SYSCAT and SYSIBM tables
   - SELECT and UPDATE on all SYSSTAT tables
   - SELECT on the following views in schema SYSIBMADM:
     - ALL_*
     - USER_*
     - ROLE_*
     - SESSION_*
     - DICTIONARY
     - TAB
3. SYSCAT.ROUTINEAUTH

   In a non-restrictive database, the special group PUBLIC is granted the following privileges:
   - EXECUTE with GRANT on all procedures in schema SQLJ
   - EXECUTE with GRANT on all functions and procedures in schema SYSFUN
   - EXECUTE with GRANT on all functions and procedures in schema SYSPROC (except audit routines)
   - EXECUTE on all table functions in schema SYSIBM
   - EXECUTE on all other procedures in schema SYSIBM
4. SYSCAT.MODULEAUTH

   In a non-restrictive database, the special group PUBLIC is granted the following privileges:
   - EXECUTE on the following modules in schema SYSIBMADM:
     - DBMS_DDL

- DBMS_JOB
- DBMS_LOB
- DBMS_OUTPUT
- DBMS_SQL
- DBMS_STANDARD
- DBMS_UTILITY

5. SYSCAT.PACKAGEAUTH
   - The database creator is granted the following privileges:
     - CONTROL on all packages created in the NULLID schema
     - BIND with GRANT on all packages created in the NULLID schema
     - EXECUTE with GRANT on all packages created in the NULLID schema
   - In a non-restrictive database, the special group PUBLIC is granted the following privileges:
     - BIND on all packages created in the NULLID schema
     - EXECUTE on all packages created in the NULLID schema
6. SYSCAT.SCHEMAAUTH

   In a non-restrictive database, the special group PUBLIC is granted the following privileges:
   - CREATEIN on schema SQLJ
   - CREATEIN on schema NULLID
7. SYSCAT.TBSPACEAUTH

   In a non-restrictive database, the special group PUBLIC is granted the USE privilege on table space USERSPACE1.
8. SYSCAT.WORKLOADAUTH

   In a non-restrictive database, the special group PUBLIC is granted the USAGE privilege on SYSDEFAULTUSERWORKLOAD.
9. SYSCAT.VARIABLEAUTH

   In a non-restrictive database, the special group PUBLIC is granted the READ privilege on schema global variables in the SYSIBM schema, execpt for the following variables:
   - SYSIBM.CLIENT_ORIGUSERID
   - SYSIBM.CLIENT_USRSECTOKEN

A non-restrictive database is a database created without the RESTRICTIVE option on the CREATE DATABASE command.

# Chapter 9. Granting privileges

To grant privileges on most database objects, you must have ACCESSCTRL authority, SECADM authority, or CONTROL privilege on that object; or, you must hold the privilege WITH GRANT OPTION. Additionally, users with SYSADM or SYSCTRL authority can grant table space privileges. You can grant privileges only on existing objects.

## About this task

To grant CONTROL privilege to someone else, you must have ACCESSCTRL or SECADM authority. To grant ACCESSCTRL, DATAACCESS, DBADM or SECADM authority, you must have SECADM authority.

The GRANT statement allows an authorized user to grant privileges. A privilege can be granted to one or more authorization names in one statement; or to PUBLIC, which makes the privileges available to all users. Note that an authorization name can be either an individual user or a group.

On operating systems where users and groups exist with the same name, you should specify whether you are granting the privilege to the user or group. Both the GRANT and REVOKE statements support the keywords USER, GROUP, and ROLE. If these optional keywords are not used, the database manager checks the operating system security facility to determine whether the authorization name identifies a user or a group; it also checks whether an authorization ID of type role with the same name exists. If the database manager cannot determine whether the authorization name refers to a user, a group, or a role, an error is returned. The following example grants SELECT privileges on the EMPLOYEE table to the user HERON:

```
GRANT SELECT
 ON EMPLOYEE TO USER HERON
```

The following example grants SELECT privileges on the EMPLOYEE table to the group HERON:

```
GRANT SELECT
   ON EMPLOYEE TO GROUP HERON
```

# Chapter 10. Revoking privileges

The REVOKE statement allows authorized users to revoke privileges previously granted to other users.

## About this task

To revoke privileges on database objects, you must have ACCESSCTRL authority, SECADM authority, or CONTROL privilege on that object. Table space privileges can also be revoked by users with SYSADM and SYSCTRL authority. Note that holding a privilege WITH GRANT OPTION is not sufficient to revoke that privilege. To revoke CONTROL privilege from another user, you must have ACCESSCTRL, or SECADM authority. To revoke ACCESSCTRL, DATAACCESS, DBADM or SECADM authority, you must have SECADM authority. Table space privileges can be revoked only by a user who holds SYSADM, or SYSCTRL authority. Privileges can only be revoked on existing objects.

**Note:** A user without ACCESSCTRL authority, SECADM authority, or CONTROL privilege is not able to revoke a privilege that they granted through their use of the WITH GRANT OPTION. Also, there is no cascade on the revoke to those who have received privileges granted by the person being revoked.
If an explicitly granted table (or view) privilege is revoked from a user with DBADM authority, privileges will not be revoked from other views defined on that table. This is because the view privileges are available through the DBADM authority and are not dependent on explicit privileges on the underlying tables.

If a privilege has been granted to a user, a group, or a role with the same name, you must specify the GROUP, USER, or ROLE keyword when revoking the privilege. The following example revokes the SELECT privilege on the EMPLOYEE table from the user HERON:

```
REVOKE SELECT
   ON EMPLOYEE FROM USER HERON
```

The following example revokes the SELECT privilege on the EMPLOYEE table from the group HERON:

```
REVOKE SELECT
   ON EMPLOYEE FROM GROUP HERON
```

Note that revoking a privilege from a group may not revoke it from all members of that group. If an individual name has been directly granted a privilege, it will keep it until that privilege is directly revoked.

If a table privilege is revoked from a user, privileges are also revoked on any view created by that user which depends on the revoked table privilege. However, only the privileges implicitly granted by the system are revoked. If a privilege on the view was granted directly by another user, the privilege is still held.

If a table privilege is revoked from a user, privileges are also revoked on any view created by that user which depends on the revoked table privilege. However, only the privileges implicitly granted by the system are revoked. If a privilege on the view was granted directly by another user, the privilege is still held.

You may have a situation where you want to GRANT a privilege to a group and then REVOKE the privilege from just one member of the group. There are only a couple of ways to do that without receiving the error message SQL0556N:

- You can remove the member from the group; or, create a new group with fewer members and GRANT the privilege to the new group.
- You can REVOKE the privilege from the group and then GRANT it to individual users (authorization IDs).

**Note:** When CONTROL privilege is revoked from a user on a table or a view, the user continues to have the ability to grant privileges to others. When given CONTROL privilege, the user also receives all other privileges WITH GRANT OPTION. Once CONTROL is revoked, all of the other privileges remain WITH GRANT OPTION until they are explicitly revoked.

All packages that are dependent on revoked privileges are marked invalid, but can be validated if rebound by a user with appropriate authority. Packages can also be rebuilt if the privileges are subsequently granted again to the binder of the application; running the application will trigger a successful implicit rebind. If privileges are revoked from PUBLIC, all packages bound by users having only been able to bind based on PUBLIC privileges are invalidated. If DBADM authority is revoked from a user, all packages bound by that user are invalidated including those associated with database utilities. Attempting to use a package that has been marked invalid causes the system to attempt to rebind the package. If this rebind attempt fails, an error occurs (SQLCODE -727). In this case, the packages must be explicitly rebound by a user with:

- Authority to rebind the packages
- Appropriate authority for the objects used within the packages

These packages should be rebound at the time the privileges are revoked.

If you define a trigger or SQL function based on one or more privileges and you lose one or more of these privileges, the trigger or SQL function cannot be used.

# Chapter 11. Controlling access to data with views

A view provides a means of controlling access or extending privileges to a table.

Using a view allows the following kinds of control over access to a table:

- Access only to designated columns of the table.

  For users and application programs that require access only to specific columns of a table, an authorized user can create a view to limit the columns addressed only to those required.
- Access only to a subset of the rows of the table.

  By specifying a WHERE clause in the subquery of a view definition, an authorized user can limit the rows addressed through a view.
- Access only to a subset of the rows or columns in data source tables or views. If you are accessing data sources through nicknames, you can create local DB2 database views that reference nicknames. These views can reference nicknames from one or many data sources.

  **Note:** Because you can create a view that contains nickname references for more than one data source, your users can access data in multiple data sources from one view. These views are called *multi-location views*. Such views are useful when joining information in columns of sensitive tables across a distributed environment or when individual users lack the privileges needed at data sources for specific objects.

To create a view, a user must have DATAACCESS authority, or CONTROL or SELECT privilege for each table, view, or nickname referenced in the view definition. The user must also be able to create an object in the schema specified for the view. That is, DBADM authority, CREATEIN privilege for an existing schema, or IMPLICIT_SCHEMA authority on the database if the schema does not already exist.

If you are creating views that reference nicknames, you do not need additional authority on the data source objects (tables and views) referenced by nicknames in the view; however, users of the view must have SELECT authority or the equivalent authorization level for the underlying data source objects when they access the view.

If your users do not have the proper authority at the data source for underlying objects (tables and views), you can:

1. Create a data source view over those columns in the data source table that are OK for the user to access
2. Grant the SELECT privilege on this view to users
3. Create a nickname to reference the view

Users can then access the columns by issuing a SELECT statement that references the new nickname.

The following scenario provides a more detailed example of how views can be used to restrict access to information.

Many people might require access to information in the STAFF table, for different reasons. For example:

- The personnel department needs to be able to update and look at the entire table.

  This requirement can be easily met by granting SELECT and UPDATE privileges on the STAFF table to the group PERSONNL:

  ```
  GRANT SELECT,UPDATE ON TABLE STAFF TO GROUP PERSONNL
  ```

- Individual department managers need to look at the salary information for their employees.

  This requirement can be met by creating a view for each department manager. For example, the following view can be created for the manager of department number 51:

  ```
  CREATE VIEW EMP051 AS
     SELECT NAME,SALARY,JOB FROM STAFF
     WHERE DEPT=51
  GRANT SELECT ON TABLE EMP051 TO JANE
  ```

  The manager with the authorization name JANE would query the EMP051 view just like the STAFF table. When accessing the EMP051 view of the STAFF table, this manager views the following information:

| NAME | SALARY | JOB |
|---|---|---|
| Fraye | 45150.0 | Mgr |
| Williams | 37156.5 | Sales |
| Smith | 35654.5 | Sales |
| Lundquist | 26369.8 | Clerk |
| Wheeler | 22460.0 | Clerk |

- All users need to be able to locate other employees. This requirement can be met by creating a view on the NAME column of the STAFF table and the LOCATION column of the ORG table, and by joining the two tables on their corresponding DEPT and DEPTNUMB columns:

  ```
  CREATE VIEW EMPLOCS AS
     SELECT NAME, LOCATION FROM STAFF, ORG
     WHERE STAFF.DEPT=ORG.DEPTNUMB
  GRANT SELECT ON TABLE EMPLOCS TO PUBLIC
  ```

  Users who access the employee location view will see the following information:

| NAME | LOCATION |
|---|---|
| Molinare | New York |
| Lu | New York |
| Daniels | New York |
| Jones | New York |
| Hanes | Boston |
| Rothman | Boston |
| Ngan | Boston |
| Kermisch | Boston |
| Sanders | Washington |
| Pernal | Washington |
| James | Washington |
| Sneider | Washington |
| Marenghi | Atlanta |

| NAME | LOCATION |
|---|---|
| O'Brien | Atlanta |
| Quigley | Atlanta |
| Naughton | Atlanta |
| Abrahams | Atlanta |
| Koonitz | Chicago |
| Plotz | Chicago |
| Yamaguchi | Chicago |
| Scoutten | Chicago |
| Fraye | Dallas |
| Williams | Dallas |
| Smith | Dallas |
| Lundquist | Dallas |
| Wheeler | Dallas |
| Lea | San Francisco |
| Wilson | San Francisco |
| Graham | San Francisco |
| Gonzales | San Francisco |
| Burke | San Francisco |
| Quill | Denver |
| Davis | Denver |
| Edwards | Denver |
| Gafney | Denver |

# Chapter 12. Roles

Roles simplify the administration and management of privileges by offering an equivalent capability as groups but without the same restrictions.

A role is a database object that groups together one or more privileges and can be assigned to users, groups, PUBLIC, or other roles by using a GRANT statement, or can be assigned to a trusted context by using a CREATE TRUSTED CONTEXT or ALTER TRUSTED CONTEXT statement. A role can be specified for the SESSION_USER ROLE connection attribute in a workload definition.

Roles provide several advantages that make it easier to manage privileges in a database system:

- Security administrators can control access to their databases in a way that mirrors the structure of their organizations (they can create roles in the database that map directly to the job functions in their organizations).
- Users are granted membership in the roles that reflect their job responsibilities. As their job responsibilities change, their membership in roles can be easily granted and revoked.
- The assignment of privileges is simplified. Instead of granting the same set of privileges to each individual user in a particular job function, the administrator can grant this set of privileges to a role representing that job function and then grant that role to each user in that job function.
- A role's privileges can be updated and all users who have been granted that role receive the update; the administrator does not need to update the privileges for every user on an individual basis.
- The privileges and authorities granted to roles are always used when you create views, triggers, materialized query tables (MQTs), static SQL and SQL routines, whereas privileges and authorities granted to groups (directly or indirectly) are not used.

  This is because the DB2 database system cannot determine when membership in a group changes, as the group is managed by third-party software (for example, the operating system or an LDAP directory). Because roles are managed inside the database, the DB2 database system can determine when authorization changes and act accordingly. Roles granted to groups are not considered, due to the same reason groups are not considered.
- All the roles assigned to a user are enabled when that user establishes a connection, so all privileges and authorities granted to roles are taken into account when a user connects. Roles cannot be explicitly enabled or disabled.
- The security administrator can delegate management of a role to others.

All DB2 privileges and authorities that can be granted within a database can be granted to a role. For example, a role can be granted any of the following authorities and privileges:

- DBADM, SECADM, DATAACCESS, ACCESSCTRL, SQLADM, WLMADM, LOAD, and IMPLICIT_SCHEMA database authorities
- CONNECT, CREATETAB, CREATE_NOT_FENCED, BINDADD, CREATE_EXTERNAL_ROUTINE, or QUIESCE_CONNECT database authorities
- Any database object privilege (including CONTROL)

A user's roles are automatically enabled and considered for authorization when a user connects to a database; you do not need to activate a role by using the SET ROLE statement. For example, when you create a view, a materialized query table (MQT), a trigger, a package, or an SQL routine, the privileges that you gain through roles apply. However, privileges that you gain through roles granted to groups of which you are a member do not apply.

A role does not have an owner. The security administrator can use the WITH ADMIN OPTION clause of the GRANT statement to delegate management of the role to another user, so that the other user can control the role membership.

### Restrictions

There are a few restrictions in the use of roles:

- A role cannot own database objects.
- Permissions and roles granted to groups are not considered when you create the following database objects:
  - Packages containing static SQL
  - Views
  - Materialized query tables (MQT)
  - Triggers
  - SQL Routines

  Only roles granted to the user creating the object or to PUBLIC, directly or indirectly (such as through a role hierarchy), are considered when creating these objects.

## Roles compared to groups

Privileges and authorities granted to groups are not considered when creating views, materialized query tables (MQTs), SQL routines, triggers, and packages containing static SQL. Avoid this restriction by using roles instead of groups.

Roles allow users to create database objects using their privileges acquired through roles, which are controlled by the DB2 database system. Groups and users are controlled externally from the DB2 database system, for example, by an operating system or an LDAP server.

### Example of replacing the use of groups with roles

This example shows how you can replace groups by using roles.

Assume that there are three groups, DEVELOPER_G, TESTER_G and SALES_G. The users BOB, ALICE, and TOM are members of these groups, as shown in the following table:

*Table 4. Example groups and users*

| Group | Users belonging to this group |
|---|---|
| DEVELOPER_G | BOB |
| TESTER_G | ALICE, TOM |
| SALES_G | ALICE, BOB |

1. The security administrator creates the roles DEVELOPER, TESTER, and SALES to be used instead of the groups.

```
CREATE ROLE DEVELOPER
CREATE ROLE TESTER
CREATE ROLE SALES
```

2. The security administrator grants membership in these roles to users (setting the membership of users in groups was the responsibility of the system administrator):

```
GRANT  ROLE DEVELOPER TO USER BOB
GRANT  ROLE TESTER TO USER ALICE, USER TOM
GRANT  ROLE SALES TO USER BOB, USER ALICE
```

3. The database administrator can grant to the roles similar privileges or authorities as were held by the groups, for example:

```
GRANT privilege ON object TO ROLE DEVELOPER
```

   The database administrator can then revoke these privileges from the groups, as well as ask the system administrator to remove the groups from the system.

## Example of creating a trigger using privileges acquired through a role

This example shows that user BOB can successfully create a trigger, TRG1, when he holds the necessary privilege through the role DEVELOPER.

1. First, user ALICE creates the table, WORKITEM:

```
CREATE TABLE WORKITEM (x int)
```

2. Then, the privilege to alter ALICE's table is granted to role DEVELOPER by the database administrator.

```
GRANT ALTER ON ALICE.WORKITEM TO ROLE DEVELOPER
```

3. User BOB successfully creates the trigger, TRG1, because he is a member of the role, DEVELOPER.

```
CREATE TRIGGER TRG1 AFTER DELETE ON ALICE.WORKITEM
           FOR EACH STATEMENT MODE DB2SQL INSERT INTO ALICE.WORKITEM VALUES (1)
```

# Chapter 13. Trusted contexts and trusted connections

A trusted context is a database object that defines a trust relationship for a connection between the database and an external entity such as an application server.

The trust relationship is based upon the following set of attributes:

- System authorization ID: Represents the user that establishes a database connection
- IP address (or domain name): Represents the host from which a database connection is established
- Data stream encryption: Represents the encryption setting (if any) for the data communication between the database server and the database client

When a user establishes a database connection, the DB2 database system checks whether the connection matches the definition of a trusted context object in the database. When a match occurs, the database connection is said to be trusted.

A trusted connection allows the initiator of this trusted connection to acquire additional capabilities that may not be available outside the scope of the trusted connection. The additional capabilities vary depending on whether the trusted connection is explicit or implicit.

The initiator of an explicit trusted connection has the ability to:

- Switch the current user ID on the connection to a different user ID with or without authentication
- Acquire additional privileges via the role inheritance feature of trusted contexts

An implicit trusted connection is a trusted connection that is not explicitly requested; the implicit trusted connection results from a normal connection request rather than an explicit trusted connection request. No application code changes are needed to obtain an implicit connection. Also, whether you obtain an implicit trusted connection or not has no effect on the connect return code (when you request an explicit trusted connection, the connect return code indicates whether the request succeeds or not). The initiator of an implicit trusted connection can only acquire additional privileges via the role inheritance feature of trusted contexts; they cannot switch the user ID.

## How using trusted contexts enhances security

The three-tiered application model extends the standard two-tiered client and server model by placing a middle tier between the client application and the database server. It has gained great popularity in recent years particularly with the emergence of web-based technologies and the Java 2 Enterprise Edition (J2EE) platform. An example of a software product that supports the three-tier application model is IBM WebSphere® Application Server (WAS).

In a three-tiered application model, the middle tier is responsible for authenticating the users running the client applications and for managing the interactions with the database server. Traditionally, all the interactions with the database server occur through a database connection established by the middle tier using a combination of a user ID and a credential that identify that middle tier to the database server. This means that the database server uses the database privileges

associated with the middle tier's user ID for all authorization checking and auditing that must occur for any database access, including access performed by the middle tier on behalf of a user.

While the three-tiered application model has many benefits, having all interactions with the database server (for example, a user request) occur under the middle tier's authorization ID raises several security concerns, which can be summarized as follows:

- Loss of user identity

  Some enterprises prefer to know the identity of the actual user accessing the database for access control purposes.
- Diminished user accountability

  Accountability through auditing is a basic principle in database security. Not knowing the user's identity makes it difficult to distinguish the transactions performed by the middle tier for its own purpose from those performed by the middle tier on behalf of a user.
- Over granting of privileges to the middle tier's authorization ID

  The middle tier's authorization ID must have all the privileges necessary to execute all the requests from all the users. This has the security issue of enabling users who do not need access to certain information to obtain access anyway.
- Weakened security

  In addition to the privilege issue raised in the previous point, the current approach requires that the authorization ID used by the middle tier to connect must be granted privileges on all resources that might be accessed by user requests. If that middle-tier authorization ID is ever compromised, then all those resources will be exposed.
- "Spill over" between users of the same connection

  Changes by a previous user can affect the current user.

Clearly, there is a need for a mechanism whereby the actual user's identity and database privileges are used for database requests performed by the middle tier on behalf of that user. The most straightforward approach of achieving this goal would be for the middle-tier to establish a new connection using the user's ID and password, and then direct the user's requests through that connection. Although simple, this approach suffers from several drawbacks which include the following:

- Inapplicability for certain middle tiers. Many middle-tier servers do not have the user authentication credentials needed to establish a connection.
- Performance overhead. There is an obvious performance overhead associated with creating a new physical connection and re-authenticating the user at the database server.
- Maintenance overhead. In situations where you are not using a centralized security set up or are not using single sign-on, there is maintenance overhead in having two user definitions (one on the middle tier and one at the server). This requires changing passwords at different places.

The trusted contexts capability addresses this problem. The security administrator can create a trusted context object in the database that defines a trust relationship between the database and the middle-tier. The middle-tier can then establish an explicit trusted connection to the database, which gives the middle tier the ability to switch the current user ID on the connection to a different user ID, with or without authentication. In addition to solving the end-user identity assertion problem, trusted contexts offer another advantage. This is the ability to control when a privilege is made available to a database user. The lack of control on when privileges are available to a user can weaken overall security. For example,

privileges may be used for purposes other than they were originally intended. The security administrator can assign one or more privileges to a role and assign that role to a trusted context object. Only trusted database connections (explicit or implicit) that match the definition of that trusted context can take advantage of the privileges associated with that role.

### Enhancing performance

When you use trusted connections, you can maximize performance because of the following advantages:

- No new connection is established when the current user ID of the connection is switched.
- If the trusted context definition does not require authentication of the user ID to switch to, then the overhead associated with authenticating a new user at the database server is not incurred.

### Example of creating a trusted context

Suppose that the security administrator creates the following trusted context object:

```
CREATE TRUSTED CONTEXT CTX1
  BASED UPON CONNECTION USING SYSTEM AUTHID USER2
  ATTRIBUTES (ADDRESS   '192.0.2.1')
  DEFAULT ROLE managerRole
   ENABLE
```

If user *user1* requests a trusted connection from IP address 192.0.2.1, the DB2 database system returns a warning (SQLSTATE 01679, SQLCODE +20360) to indicate that a trusted connection could not be established, and that user *user1* simply got a non-trusted connection. However, if user *user2* requests a trusted connection from IP address 192.0.2.1, the request is honored because the connection attributes are satisfied by the trusted context CTX1. Now that use *user2* has established a trusted connection, he or she can now acquire all the privileges and authorities associated with the trusted context role managerRole. These privileges and authorities may not be available to user *user2* outside the scope of this trusted connection

## Using trusted contexts and trusted connections

You can establish an explicit trusted connection by making a request within an application when a connection to a DB2 database is established. The security administrator must have previously defined a trusted context, using the CREATE TRUSTED CONTEXT statement, with attributes matching those of the connection you are establishing (see Step 1, later).

### Before you begin

The API you use to request an explicit trusted connection when you establish a connection depends on the type of application you are using (see the table in Step 2).

After you have established an explicit trusted connection, the application can switch the user ID of the connection to a different user ID using the appropriate API for the type of application (see the table in Step 3).

## Procedure

1. The security administrator defines a trusted context in the server by using the CREATE TRUSTED CONTEXT statement. For example:

```
CREATE TRUSTED CONTEXT MYTCX
       BASED UPON CONNECTION USING SYSTEM AUTHID NEWTON
       ATTRIBUTES (ADDRESS   '192.0.2.1')
       WITH USE FOR PUBLIC WITHOUT AUTHENTICATION
       ENABLE
```

2. To establish a trusted connection, use one of the following APIs in your application:

| Option | Description |
|---|---|
| **Application** | **API** |
| **CLI/ODBC** | SQLConnect, SQLSetConnectAttr |
| **XA CLI/ODBC** | Xa_open |
| **JAVA** | getDB2TrustedPooledConnection, getDB2TrustedXAConnection |

3. To switch to a different user, with or without authentication, use one of the following APIs in your application:

| Option | Description |
|---|---|
| **Application** | **API** |
| **CLI/ODBC** | SQLSetConnectAttr |
| **XA CLI/ODBC** | SQLSetConnectAttr |
| **JAVA** | getDB2Connection, reuseDB2Connection |
| **.NET** | DB2Connection.ConnectionString keywords: TrustedContextSystemUserID and TrustedContextSystemPassword |

The switching can be done either with or without authenticating the new user ID, depending on the definition of the trusted context object associated with the explicit trusted connection. For example, suppose that the security administrator creates the following trusted context object:

```
CREATE TRUSTED CONTEXT CTX1
 BASED UPON CONNECTION USING SYSTEM AUTHID USER1
 ATTRIBUTES (ADDRESS   '192.0.2.1')
  WITH USE FOR USER2 WITH AUTHENTICATION,
               USER3 WITHOUT AUTHENTICATION
  ENABLE
```

Further, suppose that an explicit trusted connection is established. A request to switch the user ID on the trusted connection to USER3 without providing authentication information is allowed because USER3 is defined as a user of trusted context CTX1 for whom authentication is not required. However, a request to switch the user ID on the trusted connection to USER2 without providing authentication information will fail because USER2 is defined as a user of trusted context CTX1 for whom authentication information must be provided.

## Example of establishing an explicit trusted connection and switching the user

In the following example, a middle-tier server needs to issue some database requests on behalf of an end-user, but does not have access to the end-user's credentials to establish a database connection on behalf of that end-user.

You can create a trusted context object on the database server that allows the middle-tier server to establish an explicit trusted connection to the database. After establishing an explicit trusted connection, the middle-tier server can switch the current user ID of the connection to a new user ID without the need to authenticate the new user ID at the database server. The following CLI code snippet demonstrates how to establish a trusted connection using the trusted context, MYTCX, defined in Step 1, earlier, and how to switch the user on the trusted connection without authentication.

```
int main(int argc, char *argv[])
{
 SQLHANDLE henv;         /* environment handle */
 SQLHANDLE hdbc1;       /* connection handle */
 char origUserid[10] = "newton";
  char password[10] = "test";
   char switchUserid[10] = "zurbie";
 char dbName[10] = "testdb";

// Allocate the handles
SQLAllocHandle( SQL_HANDLE_ENV, &henv );
SQLAllocHandle( SQL_HANDLE_DBC, &hdbc1 );

// Set the trusted connection attribute
SQLSetConnectAttr( hdbc1, SQL_ATTR_USE_TRUSTED_CONTEXT,
SQL_TRUE, SQL_IS_INTEGER );

// Establish a trusted connection
SQLConnect( hdbc1, dbName, SQL_NTS, origUserid, SQL_NTS,
password, SQL_NTS );

//Perform some work under user ID "newton"
. . . . . .

 // Commit the work
SQLEndTran(SQL_HANDLE_DBC, hdbc1, SQL_COMMIT);


// Switch the user ID on the trusted connection
SQLSetConnectAttr( hdbc1,
SQL_ATTR_TRUSTED_CONTEXT_USERID, switchUserid,
SQL_IS_POINTER
);

//Perform new work using user ID "zurbie"
. . . . . .

//Commit the work
SQLEndTranSQL_HANDLE_DBC, hdbc1, SQL_COMMIT);

// Disconnect from database
SQLDisconnect( hdbc1 );

 return 0;

}  /* end of main */
```

## What to do next

**When does the user ID actually get switched?**

After the command to switch the user on the trusted connection is issued, the switch user request is not performed until the next statement is sent to the server. This is demonstrated by the following example where the **`list applications`** command shows the original user ID until the next statement is issued.

1. Establish an explicit trusted connection with USERID1.
2. Issue the switch user command, such as **`getDB2Connection`** for USERID2.
3. Run `db2 list applications`. It still shows that USERID1 is connected.
4. Issue a statement on the trusted connection, such as `executeQuery("values current sqlid")`, to perform the switch user request at the server.
5. Run `db2 list applications` again. It now shows that USERID2 is connected.

# Chapter 14. Row and column access control (RCAC)

DB2 Version 10.1 introduces row and column access control (RCAC), as an additional layer of data security. Row and column access control is sometimes referred to as fine-grained access control or FGAC. RCAC controls access to a table at the row level, column level, or both. RCAC can be used to complement the table privileges model.

To comply with various government regulations, you might implement procedures and methods to ensure that information is adequately protected. Individuals in your organization are permitted access to only the subset of data that is required to perform their job tasks. For example, government regulations in your area might state that a doctor is authorized to view the medical records of their own patients, but not of other patients. The same regulations might also state that, unless a patient gives their consent, a healthcare provider is not permitted access to patient personal information, such as the patients home phone number.

You can use row and column access control to ensure that your users have access to only the data that is required for their work. For example, a hospital system running DB2 for Linux, UNIX, and Windows and RCAC can filter patient information and data to include only that data which a particular doctor requires. Other patients do not exist as far as the doctor is concerned. Similarly, when a patient service representative queries the patient table at the same hospital, they are able to view the patient name and telephone number columns, but the medical history column is masked for them. If data is masked, a NULL, or an alternate value is displayed, instead of the actual medical history.

Row and column access control, or RCAC, has the following advantages:

- No database user is inherently exempted from the row and column access control rules.

  Even higher level authorities such as users with DATAACCESS authority are not exempt from these rules. Only users with security administrator (SECADM) authority can manage row and column access controls within a database. Therefore, you can use RCAC to prevent users with DATAACCESS authority from freely accessing all data in a database.
- Table data is protected regardless of how a table is accessed via SQL.

  Applications, improvised query tools, and report generation tools are all subject to RCAC rules. The enforcement is data-centric.
- No application changes are required to take advantage of this additional layer of data security.

  That is, row and column level access controls are established and defined in a way that is not apparent to existing applications. However, RCAC represents an important shift in paradigm in the sense that it is no longer what is being asked but rather who is asking what. Result sets for the same query change based on the context in which the query was asked and there is no warning or error returned. This behavior is the exact intent of the solution. It means that application designers and DBAs must be conscious that queries do not see the whole picture in terms of the data in the table, unless granted specific permissions to do so.

## Row and column access control (RCAC) rules

Row and column access control (RCAC) places access control at the table level around the data itself. SQL rules created on rows and columns are the basis of the implementation of this capability.

Row and column access control is an access control model in which a security administrator manages privacy and security policies. RCAC permits all users to access the same table, as opposed to alternative views of a table. RCAC does however, restrict access to the table based upon individual user permissions or rules as specified by a policy associated with the table. There are two sets of rules, one set operates on rows, and the other on columns.

- Row permission
  - A row permission is a database object that expresses a row access control rule for a specific table.
  - A row access control rule is an SQL search condition that describes what set of rows a user has access to.
- Column mask
  - A column mask is a database object that expresses a column access control rule for a specific column in a table.
  - A column access control rule is an SQL CASE expression that describes what column values a user is permitted to see and under what conditions.

## Scenario: ExampleHMO using row and column access control

This scenario presents ExampleHMO, a national organization with a large and active list of patients, as a user of row and column access control. ExampleHMO uses row and column access control to ensure that their database policies reflect government regulatory requirements for privacy and security, as well as management business objectives.

Organizations that handle patient health information and their personal information, like ExampleHMO, must comply with government privacy and data protection regulations, for example the Health Insurance Portability and Accountability Act (HIPAA). These privacy and data protection regulations ensure that any sensitive patient medical or personal information is shared, viewed, and modified only by authorities who are privileged to do so. Any violation of the act results in huge penalties including civil and criminal suits.

ExampleHMO must ensure that the data stored in their database systems is secure and only privileged users have access to the data. According to typical privacy regulations, certain patient information can be accessed and modified by only privileged users.

### Security policies

ExampleHMO implements a security strategy where data access to DB2 databases are made available according to certain security policies.

The security policies conform to government privacy and data protection regulations. The first column outlines the policies and the challenges faced by the organization, the second column outlines the DB2 row and column access control feature which addresses the challenge.

| Security challenge | Row and column access control feature which addresses the security challenge |
| --- | --- |
| Limiting column access to only privileged users.<br><br>For example, Jane, who is a drug researcher at a partner company, is not permitted to view sensitive patient medical information or personal data like their insurance number. | Column masks can be used to filter or hide sensitive data from Jane. |
| Limiting row access to only privileged users. Dr. Lee is only permitted to view patient information for his own patients, not all patients in the ExampleHMO system. | Row permissions can be implemented to control which user can view any particular row. |
| Restricting data on a need-to-know basis. | Row permissions can help with this challenge as well by restricting table level data at the user level. |
| Restricting other database objects like UDFs, triggers, views on RCAC secured data. | Row and column access control protects data at the data level. It is this data-centric nature of the row and column access control solution that enforces security policies on even database objects like UDFs, triggers, and views. |

## Database users and roles

In this scenario, a number of different people create, secure, and use ExampleHMO data. These people have different user rights and database authorities.

ExampleHMO implemented their security strategy to classify the way data is accessed from the database. Internal and external access to data is based on the separation of duties to users who access the data and their data access privileges. ExampleHMO created the following database roles to separate these duties:

**PCP**
For primary care physicians.

**DRUG_RESEARCH**
For researchers.

**ACCOUNTING**
For accountants.

**MEMBERSHIP**
For members who add patients for opt-in and opt-out.

**PATIENT**
For patients.

The following people create, secure, and use ExampleHMO data:

**Alex**
ExampleHMO Chief Security Administrator. He holds the SECADM authority.

**Peter**
ExampleHMO Database Administrator. He holds the DBADM authority.

**Paul**
ExampleHMO Database Developer. He has the privileges to create triggers and user-defined functions.

**Dr. Lee**
ExampleHMO Physician. He belongs to the PCP role.

**Jane**
Drug researcher at Innovative Pharmaceutical Company, a ExampleHMO partner. She belongs to the DRUG_RESEARCH role.

**John**
ExampleHMO Accounting Department. He belongs to the ACCOUNTING role.

**Tom**
ExampleHMO Membership Officer. He belongs to the MEMBERSHIP role.

**Bob**
ExampleHMO Patient. He belongs to the PATIENT role.

If you want to try any of the example SQL statements and commands presented in this scenario, create these user IDs with their listed authorities.

The following example SQL statements assume that the users have been created on the system. The SQL statements create each role and grant **SELECT** and **INSERT** permissions to the various tables in the ExampleHMO database to the users:

```
--Creating roles and granting authority

CREATE ROLE PCP;

CREATE ROLE DRUG_RESEARCH;

CREATE ROLE ACCOUNTING;

CREATE ROLE MEMBERSHIP;

CREATE ROLE PATIENT;

GRANT ROLE PCP TO USER LEE;
GRANT ROLE DRUG_RESEARCH TO USER JANE;
GRANT ROLE ACCOUNTING  TO USER JOHN;
GRANT ROLE MEMBERSHIP TO USER TOM;
GRANT ROLE PATIENT TO USER BOB;
```

## Database tables

This scenario focuses on two tables in the ExampleHMO database: the PATIENT table and the PATIENTCHOICE table.

The PATIENT table stores basic patient information and health information. This scenario considers the following columns within the PATIENT table:

**SSN**
The patient's insurance number. A patient's insurance number is considered personal information.

**NAME**
The patient's name. A patient's name is considered personal information.

**ADDRESS**
The patient's address. A patient's address is considered personal information.

**USERID**
The patient's database ID.

**PHARMACY**
The patient's medical information.

**ACCT_BALANCE**
The patient's billing information.

**PCP_ID**
The patient's primary care physician database ID

The PATIENTCHOICE table stores individual patient opt-in and opt-out information which decides whether a patient wants to expose his health information to outsiders for research purposes in this table. This scenario considers the following columns within the PATIENTCHOICE table:

**SSN**
The patient's insurance number is used to match patients with their choices.

**CHOICE**
The name of a choice a patient can make.

**VALUE**
The decision made by the patients about the choice.

For example, the row 123-45-6789, drug_research, opt-in says that patient with SSN 123-45-6789 agrees to disclose their information for medical research purposes.

The following example SQL statements create the PATIENT, PATIENTCHOICE, and ACCT_HISTORY tables. Authority is granted on the tables and data is inserted:

```
--Patient table storing information regarding patient
CREATE TABLE PATIENT (
 SSN CHAR(11),
 USERID VARCHAR(18),
 NAME VARCHAR(128),
 ADDRESS VARCHAR(128),
 PHARMACY VARCHAR(250),
 ACCT_BALANCE DECIMAL(12,2) WITH DEFAULT,
 PCP_ID VARCHAR(18)
 );


--Patientchoice table which stores what patient opts
--to expose regarding his health information

CREATE TABLE PATIENTCHOICE (
 SSN CHAR(11),
 CHOICE VARCHAR(128),
 VALUE VARCHAR(128)
);

--Log table to track account balance
CREATE TABLE ACCT_HISTORY(
 SSN   CHAR(11),
 BEFORE_BALANCE DECIMAL(12,2),
 AFTER_BALANCE DECIMAL(12,2),
 WHEN  DATE,
 BY_WHO VARCHAR(20)
);


--Grant authority

GRANT SELECT, UPDATE ON TABLE PATIENT TO ROLE PCP;

GRANT SELECT ON TABLE PATIENT TO ROLE DRUG_RESEARCH;

GRANT SELECT, UPDATE ON TABLE PATIENT TO ROLE ACCOUNTING;
```

```
GRANT SELECT ON TABLE ACCT_HISTORY TO ROLE ACCOUNTING;

GRANT SELECT, UPDATE, INSERT ON TABLE PATIENT TO ROLE MEMBERSHIP;
GRANT INSERT ON TABLE PATIENTCHOICE TO ROLE MEMBERSHIP;

GRANT SELECT ON TABLE PATIENT TO ROLE PATIENT;

GRANT SELECT, ALTER ON TABLE PATIENT TO USER ALEX;

GRANT ALTER, SELECT ON TABLE PATIENT TO USER PAUL;
GRANT INSERT ON TABLE ACCT_HISTORY TO USER PAUL;

--Insert patient data

INSERT INTO PATIENT
  VALUES('123-55-1234', 'MAX', 'Max', 'First Strt', 'hypertension', 89.70,'LEE');
INSERT INTO PATIENTCHOICE
  VALUES('123-55-1234', 'drug-research', 'opt-out');


INSERT INTO PATIENT
  VALUES('123-58-9812', 'MIKE', 'Mike', 'Long Strt', null, 8.30,'JAMES');
INSERT INTO PATIENTCHOICE
  VALUES('123-58-9812', 'drug-research', 'opt-out');


INSERT INTO PATIENT
  VALUES('123-11-9856', 'SAM', 'Sam', 'Big Strt', null, 0.00,'LEE');
INSERT INTO PATIENTCHOICE
  VALUES('123-11-9856', 'drug-research', 'opt-in');


INSERT INTO PATIENT
  VALUES('123-19-1454', 'DUG', 'Dug', 'Good Strt', null, 0.00,'JAMES');
INSERT INTO PATIENTCHOICE
  VALUES('123-19-1454', 'drug-research', 'opt-in');
```

## Security administration

Security administration and the security administrator (SECADM) role play important parts in securing patient and company data at ExampleHMO. At ExampleHMO, management decided that different people hold database administration authority and security administration authority.

The management team at ExampleHMO decides to create a role for administering access to their data. The team also decides that even users with DATAACCESS authority are not able to view protected health and personal data by default.

The management team selects Alex to be the sole security administrator for ExampleHMO. From now on, Alex controls all data access authority. With this authority, Alex defines security rules such as row permissions, column masks, and whether functions and triggers are secure or not. These rules control which users have access to any given data under his control.

After Peter, the database administrator, creates the required tables and sets up the required roles, duties are separated. The database administration and security administration duties are separated by making Alex the security administrator.

Peter connects to the database and grants Alex SECADM authority. Peter can grant SECADM authority since he currently holds the DBADM, DATAACCESS, and SECADM authorities.

```
-- To seperate duties of security administrator from system administrator,
-- the SECADMN Peter grants SECADM authority to user Alex.

GRANT SECADM ON DATABASE TO USER ALEX;
```

Alex, after receiving the SECADM authority, connects to the database and revokes the security administrator privilege from Peter. The duties are now separated and Alex becomes the sole authority to grant data access to others within and outside ExampleHMO. The following SQL statement shows how Alex revoked SECADM authority from Peter:

```
--revokes the SECADMIN authority for Peter

REVOKE SECADM ON DATABASE FROM USER PETER;
```

## Row permissions

Alex, the security administrator, starts to restrict data access on the ExampleHMO database by using row permissions, a part of row and column access control. Row permissions filter the data returned to users by row.

Patients are permitted to view their own data. A physician is permitted to view the data of all his patients, but not the data of patients who see other physicians. Users belonging to the MEMBERSHIP, ACCOUNTING, or DRUG_RESEARCH roles can access all patient information. Alex, the security administrator, is asked to implement these permissions to restrict who can see any given row on a need-to-know basis.

Row permissions restrict or filter rows based on the user who has logged on to the database. At ExampleHMO, the row permissions create a horizontal data restriction on the table named PATIENT.

Alex implements the following row permissions so that a user in each role is restricted to view a result set that they are privileged to view:

```
CREATE PERMISSION ROW_ACCESS ON PATIENT
-------------------------------------------------------
-- Accounting information:
-- ROLE PATIENT is allowed to access his or her own row
-- ROLE PCP is allowed to access his or her patients' rows
-- ROLE MEMBERSHIP, ACCOUNTING, and DRUG_RESEARCH are
-- allowed to access all rows
-------------------------------------------------------------
FOR ROWS WHERE(VERIFY_ROLE_FOR_USER(SESSION_USER,'PATIENT') = 1
AND
PATIENT.USERID = SESSION_USER) OR
(VERIFY_ROLE_FOR_USER(SESSION_USER,'PCP') = 1
AND
PATIENT.PCP_ID = SESSION_USER) OR
 (VERIFY_ROLE_FOR_USER(SESSION_USER,'MEMBERSHIP') = 1 OR
 VERIFY_ROLE_FOR_USER(SESSION_USER,'ACCOUNTING') = 1 OR
 VERIFY_ROLE_FOR_USER(SESSION_USER, 'DRUG_RESEARCH') = 1)
ENFORCED FOR ALL ACCESS
ENABLE;
```

Alex observes that even after creating a row permission, all data can still be viewed by the other employees. A row permission is not applied until it is activated on the table for which it was defined. Alex must now activate the permission:

```
--Activate row access control to implement row permissions

ALTER TABLE PATIENT ACTIVATE ROW ACCESS CONTROL;
```

## Column masks

Alex, the security administrator, further restricts data access on the ExampleHMO database by using column masks, a part of row and column access control. Column masks hide data returned to users by column unless they are permitted to view the data.

Patient payment details must only be accessible to the users in the accounts department. The account balance must not be seen by any other database users. Alex is asked to prevent access by anyone other than users belonging to the ACCOUNTING role.

Alex implements the following column mask so that a user in each role is restricted to view a result set that they are privileged to view:

```
--Create a Column MASK ON ACCT_BALANCE column on the PATIENT table

CREATE MASK ACCT_BALANCE_MASK ON PATIENT FOR
-------------------------------------------------------------
-- Accounting information:
-- Role ACCOUNTING is allowed to access the full information
-- on column ACCT_BALANCE.
-- Other roles accessing this column will strictly view a
-- zero value.
-------------------------------------------------------------
COLUMN ACCT_BALANCE RETURN
 CASE WHEN VERIFY_ROLE_FOR_USER(SESSION_USER,'ACCOUNTING') = 1
   THEN ACCT_BALANCE
  ELSE 0.00
 END
ENABLE;
```

Alex observes that even after creating a column mask, the data can still be viewed by the other employees. A column mask is not applied until it is activated on the table for which it was defined. Alex must now activate the mask:

```
--Activate column access control to implement column masks

ALTER TABLE PATIENT ACTIVATE COLUMN ACCESS CONTROL;
```

Alex is asked by management to hide the insurance number of the patients. Only a patient, physician, accountant, or people in the MEMBERSHIP role can view the SSN column.

Also, to protect the PHARMACY detail of a patient, the information in the PHARMACY column must only be viewed by a drug researcher or a physician. Drug researchers can see the data only if the patient has agreed to disclose the information.

Alex implements the following column masks so that a user in each role is restricted to view a result set that they are privileged to view:

```
CREATE MASK SSN_MASK ON PATIENT FOR
-----------------------------------------------------
-- Personal contact information:
-- Roles PATIENT, PCP, MEMBERSHIP, and ACCOUNTING are allowed
-- to access the full information on columns SSN, USERID, NAME,
-- and ADDRESS. Other roles accessing these columns will
-- strictly view a masked value.
-----------------------------------------------------
COLUMN SSN RETURN
 CASE WHEN
  VERIFY_ROLE_FOR_USER(SESSION_USER,'PATIENT') = 1 OR
  VERIFY_ROLE_FOR_USER(SESSION_USER,'PCP') = 1 OR
```

```
  VERIFY_ROLE_FOR_USER(SESSION_USER,'MEMBERSHIP') = 1 OR
  VERIFY_ROLE_FOR_USER(SESSION_USER,'ACCOUNTING') = 1
 THEN SSN
  ELSE CHAR('XXX-XX-' || SUBSTR(SSN,8,4)) END
ENABLE;

CREATE MASK PHARMACY_MASK ON PATIENT FOR
-----------------------------------------------------
-- Medical information:
-- Role PCP is allowed to access the full information on
-- column PHARMACY.
-- For the purposes of drug research, Role DRUG_RESEARCH can
-- conditionally see a patient's medical information
-- provided that the patient has opted-in.
-- In all other cases, null values are rendered as column
-- values.
-----------------------------------------------------
COLUMN PHARMACY RETURN
  CASE WHEN
  VERIFY_ROLE_FOR_USER(SESSION_USER,'PCP') = 1 OR
  (VERIFY_ROLE_FOR_USER(SESSION_USER,'DRUG_RESEARCH')=1
  AND
   EXISTS (SELECT 1 FROM PATIENTCHOICE C
 WHERE PATIENT.SSN = C.SSN AND C.CHOICE = 'drug-research' AND C.VALUE = 'opt-in'))
  THEN PHARMACY
  ELSE NULL
END
ENABLE;
```

Alex observes that after creating these two column masks that the data is only viewable to the intended users. The PATIENT table already had column access control activated.

## Inserting data

When a new patient is admitted for treatment in the hospital, the new patient record must be added to the ExampleHMO database.

Bob is a new patient, and his records must be added to the ExampleHMO database. A user with the required security authority must create the new record for Bob. Tom, from the ExampleHMO membership department, with the MEMBERSHIP role, enrolls Bob as a new member. After connecting to the ExampleHMO database, Tom runs the following SQL statements to add Bob to the ExampleHMO database:

```
INSERT INTO PATIENT
  VALUES('123-45-6789', 'BOB', 'Bob', '123 Some St.', 'hypertension', 9.00,'LEE');
INSERT INTO PATIENTCHOICE
  VALUES('123-45-6789', 'drug-research', 'opt-in');
```

Tom confirmed that Bob was added to the database by querying the same from the PATIENT table in the ExampleHMO database:

```
Select * FROM PATIENT WHERE NAME = 'Bob';

SSN         USERID    NAME    ADDRESS       PHARMACY     ACCT_BALANCE  PCP_ID

----------- --------- ------- ------------- ------------ ------------- ------
123-45-6789 BOB       Bob     123 Some St.  XXXXXXXXXXX  0.00          LEE
```

## Updating data

While in the hospital, Bob gets his treatment changed. As a result his records in the ExampleHMO database need updating.

Dr. Lee, who is Bob's physician, advises a treatment change and changes Bob's medicine. Bob's record in the ExampleHMO systems must be updated. The row permission rules set in the ExampleHMO database specify that anyone who cannot view the data in a row cannot update the data in that row. Since Bob's PCPID contains Dr. Lee's ID, and the row permission is set, Dr. Lee can both view, and update Bob's record using the following example SQL statement:

```
UPDATE PATIENT SET PHARMACY = 'codeine' WHERE NAME = 'Bob';
```

Dr. Lee checks the update:

```
Select * FROM PATIENT WHERE NAME = 'Bob';

SSN         USERID    NAME    ADDRESS      PHARMACY    ACCT_BALANCE   PCP_ID

----------- --------- ------- ------------ ----------- -------------- ------
123-45-6789 BOB       Bob     123 Some St. codeine     0.00           LEE
```

Dug is a patient who is under the care of Dr. James, one of Dr. Lee's colleagues. Dr. Lee attempts the same update on the record for Dug:

```
UPDATE PATIENT SET PHARMACY = 'codeine' WHERE NAME = 'Dug';
SQL0100W  No row was found for FETCH, UPDATE or DELETE; or the result of a query
 is an empty table. SQLSTATE=02000
```

Since Dug's PCPID does not contain Dr. Lee's ID, and the row permission is set, Dr. Lee cannot view, or update Dug's record.

## Reading data

With row and column access control, people in different roles can have different result sets from the same database queries. For example, Peter, the database administrator with DATAACCESS authority, cannot see any data on the PATIENT table.

Peter, Bob, Dr. Lee, Tom, Jane, and John each connect to the database and try the following SQL query:

```
SELECT SSN, USERID, NAME, ADDRESS, PHARMACY, ACCT_BALANCE, PCP_ID FROM PATIENT;
```

Results of the query vary according to who runs the query. The row and column access control rules created by Alex are applied on these queries.

Here is the result set Peter sees:

```
SSN         USERID    NAME      ADDRESS     PHARMACY    ACC_BALANCE PCP_ID

----------- --------- --------- ----------- ----------- ----------- -----------

  0 record(s) selected.
```

Even though there is data in the table and Peter is the database administrator, he lacks the authority to see all data.

Here is the result set Bob sees:

```
SSN         USERID    NAME     ADDRESS     PHARMACY    ACC_BALANCE PCP_ID

----------- --------- --------- ----------- ----------- ----------- ------
123-45-6789 BOB       Bob      123 Some St.XXXXXXXXXXX 0.00        LEE

1 record(s) selected.
```

Bob, being a patient, can only see his own data. Bob belongs to the PATIENT role. The PHARMACY and ACC_BALANCE column data have been hidden from him.

Here is the result set Dr. Lee sees:

```
SSN         USERID   NAME     ADDRESS     PHARMACY            ACC_BALANCE PCP_ID

----------- -------- -------- ----------- ------------------- ----------- ------
123-55-1234 MAX      Max      First Strt  hypertension        0.00        LEE
123-11-9856 SAM      Sam      Big Strt    High blood pressure 0.00        LEE
123-45-6789 BOB      Bob      123 Some St.codeine             0.00        LEE

3 record(s) selected.
```

Dr. Lee can see only the data for patients under his care. Dr. Lee belongs to the PCP role. The ACC_BALANCE column data is hidden from him.

Here is the result set Tom sees:

```
SSN         USERID   NAME     ADDRESS     PHARMACY    ACC_BALANCE PCP_ID

----------- -------- -------- ----------- ----------- ----------- -----------
123-55-1234 MAX      Max      First Strt  XXXXXXXXXX 0.00        LEE
123-58-9812 MIKE     Mike     Long Strt   XXXXXXXXXX 0.00        JAMES
123-11-9856 SAM      Sam      Big Strt    XXXXXXXXXX 0.00        LEE
123-19-1454 DUG      Dug      Good Strt   XXXXXXXXXX 0.00        JAMES
123-45-6789 BOB      Bob      123 Some St.XXXXXXXXXX 0.00        LEE

5 record(s) selected.
```

Tom can see all members. Tom belongs to the membership role. He is not privileged to see any data in the PHARMACY and ACC_BALANCE columns.

Here is the result set Jane sees:

```
SSN         USERID   NAME     ADDRESS     PHARMACY            ACC_BALANCE PCP_ID

----------- -------- -------- ----------- ------------------- ----------- -------
XXX-XX-1234 MAX      Max      First Strt  XXXXXXXXXX          0.00        LEE
XXX-XX-9812 MIKE     Mike     Long Strt   XXXXXXXXXX          0.00        JAMES
XXX-XX-9856 SAM      Sam      Big Strt    High blood pressure 0.00        LEE
XXX-XX-1454 DUG      Dug      Good Strt   Influenza           0.00        JAMES
XXX-XX-6789 BOB      Bob      123 Some St.codeine             0.00        LEE

5 record(s) selected.
```

Jane can see all members. She belongs to the DRUG_RESEARCH role. The SSN and ACC_BALANCE column data are hidden from her. The PHARMACY data is only available if the patients have opted-in to share their data with drug research companies.

Here is the result set John sees:

```
SSN         USERID   NAME     ADDRESS     PHARMACY    ACC_BALANCE PCP_ID

----------- -------- -------- ----------- ----------- ----------- --------
123-55-1234 MAX      Max      First Strt  XXXXXXXXXX 89.70       LEE
123-58-9812 MIKE     Mike     Long Strt   XXXXXXXXXX 8.30        JAMES
123-11-9856 SAM      Sam      Big Strt    XXXXXXXXXX 0.00        LEE
123-19-1454 DUG      Dug      Good Strt   XXXXXXXXXX 0.00        JAMES
123-45-6789 BOB      Bob      123 Some St.XXXXXXXXXX 9.00        LEE

5 record(s) selected.
```

John can see all members. He belongs to the ACCOUNTING role. The PHARMACY column data is hidden from him.

## Revoking authority

Alex, as security administrator, is responsible for controlling who can create secure objects. When developers are done creating secure objects, Alex revokes their authority on the database.

Paul, the database developer, is done with development activities. Alex immediately revokes the create authority from Paul:

```
REVOKE CREATE_SECURE_OBJECT ON DATABASE FROM USER PAUL;
```

If Paul must create secure objects in the future, he must speak to Alex to have the create authority granted again.

# Part 3. Working with Databases and Database Objects

To start working with DB2 databases, you must create instances, databases, and database objects.

You can create the following database objects in a DB2 database:

- Tables
- Constraints
- Indexes
- Triggers
- Sequences
- Views
- Usage lists

You ca use Data Definition Language (DDL) statements or tools such as IBM Data Studio to create these database objects. The DDL statements are generally prefixed by the keywords CREATE or ALTER.

Understanding the features and functionality that each of these database objects provides is important to implement a good database design that meets your current business's data storage needs while remaining flexible enough to accommodate expansion and growth over time.

# Chapter 15. Instances

DB2 databases are created within DB2 instances on the database server. The creation of multiple instances on the same physical server provides a unique database server environment for each instance.

For example, you can maintain a test environment and a production environment on the same computer, or you can create an instance for each application and then fine-tune each instance specifically for the application it will service, or, to protect sensitive data, you can have your payroll database stored in its own instance so that owners of other instances (on the same server) cannot see payroll data.

The installation process creates a default DB2 instance, which is defined by the DB2INSTANCE environment variable. This is the instance that is used for most operations. However, instances can be created (or dropped) after installation.

When determining and designing the instances for your environment, note that each instance controls access to one or more databases. Every database within an instance is assigned a unique name, has its own set of system catalog tables (which are used to keep track of objects that are created within the database), and has its own configuration file. Each database also has its own set of grantable authorities and privileges that govern how users interact with the data and database objects stored in it. Figure 4 shows the hierarchical relationship among systems, instances, and databases.

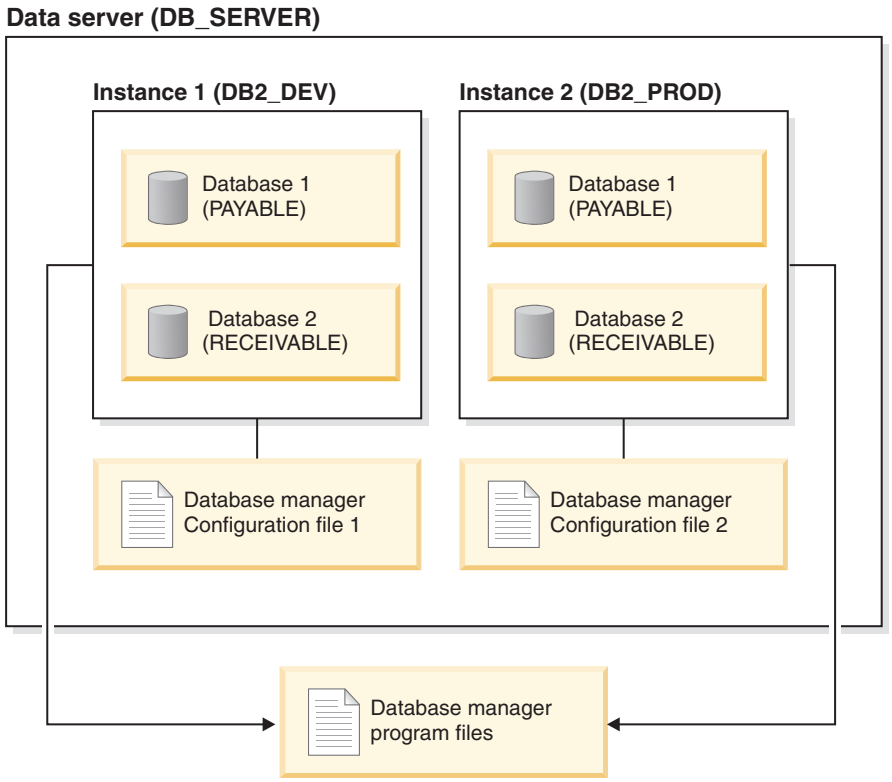



*Figure 4. Hierarchical relationship among DB2 systems, instances, and databases*

You also must be aware of another particular type of instance called the *DB2 administration server* (DAS). The DAS is a special DB2 administration control point used to assist with the administration tasks only on other DB2 servers. A DAS must be running if you want to use the Client Configuration Assistant to discover the remote databases or the graphical tools that come with the DB2 product, for example, the IBM Data Studio. There is only one DAS in a DB2 database server, even when there are multiple instances.

**Important:** The DB2 Administration Server (DAS) has been deprecated in Version 9.7 and might be removed in a future release. The DAS is not supported in DB2 pureScale environments. Use software programs that use the Secure Shell protocol for remote administration. For more information, see " DB2 administration server (DAS) has been deprecated" at .

Once your instances are created, you can attach to any other instance available (including instances on other systems). Once attached, you can perform maintenance and utility tasks that can only be done at the instance level, for example, create a database, force applications off a database, monitor database activity, or change the contents of the database manager configuration file that is associated with that particular instance.

# Chapter 16. Databases

A DB2 database is a *relational database*. The *database* stores all data in tables that are related to one another. Relationships are established between tables such that data is shared and duplication is minimized.

A *relational database* is a database that is treated as a set of tables and manipulated in accordance with the relational model of data. It contains a set of objects used to store, manage, and access data. Examples of such objects are tables, views, indexes, functions, triggers, and packages. Objects can be either defined by the system (built-in objects) or defined by the user (user-defined objects).

A *distributed relational database* consists of a set of tables and other objects that are spread across different but interconnected computer systems. Each computer system has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given database manager to execute SQL statements on another computer system.

A *partitioned relational database* is a relational database whose data is managed across multiple database partitions. This separation of data across database partitions is transparent to most SQL statements. However, some data definition language (DDL) statements take database partition information into consideration (for example, **CREATE DATABASE PARTITION GROUP**). DDL is the subset of SQL statements used to describe data relationships in a database.

A *federated database* is a relational database whose data is stored in multiple data sources (such as separate relational databases). The data appears as if it were all in a single large database and can be accessed through traditional SQL queries. Changes to the data can be explicitly directed to the appropriate data source.

## Database directories and files

When you create a database, information about the database including default information is stored in a directory hierarchy.

The hierarchical directory structure is created for you. You can specify the location of the structure by specifying a directory path or drive for the **CREATE DATABASE** command; if you do not specify a location, a default location is used.

In the directory that you specify as the database path in the **CREATE DATABASE** command, a subdirectory that uses the name of the instance is created.

Within the instance-name subdirectory, the *partition-global directory* is created. The partition-global directory contains global information associated with your new database. The partition-global directory is named `NODE`*xxxx*`/SQL`*yyyyy*, where *xxxx* is the data partition number and *yyyyy* is the database token (numbered >=1).

Under the *partition-global directory*, the *member-specific* directory is created. The member-specific directory contains local database information. The member-specific directory is named `MEMBER`*xxxx* where *xxxx* is the member number.

- In a DB2 pureScale environment, there is a member-specific directory for each member, called `MEMBER0000`, `MEMBER0001`, and so on.

- In a partitioned database environment, member numbers have a one-to-one mapping with their corresponding partition number, therefore there is one NODE*xxxx* directory per member and partition. Member-specific directories are always named MEMBERxxxx and they always reside under the partition-global directory.
- An Enterprise Server Edition environment runs on a single member, and has one member-specific directory, called MEMBER0000.

## Partition-global directory

The partition-global directory has the path: *your_instance*/*NODExxxx*/SQLxxxxx.

The partition-global directory contains the following files:

- Global deadlock write-to-file event monitor files that specify either a relative path or no path at all.
- Table space information files.

  The files SQLSPCS.1 and SQLSPCS.2 contain table space information. These files are duplicates of each other for backup purposes.
- Storage group control files.

  The files SQLSGF.1 and SQLSGF.2 contain storage group information associated with the automatic storage feature of a database. These files are duplicates of each other for maintenance and backup purposes. The files are created for a database when you create the database using the **CREATE DATABASE** command or when you upgrade a nonautomatic storage database to DB2 Version 10.1 or later.
- Temporary table space container files.

  The default directory for new containers is instance/NODExxxx/<db-name>. The files are managed locally by each member. The table space file names are made unique for each member by inserting the member number into the file name, for example: /*storage path*/SAMPLEDB/T0000011/C0000000.TMP/SQL00002.MEMBER0001.TDA
- The global configuration file.

  The global configuration file, SQLDBCONF, contains database configuration parameters that refer to single, shared resources that must remain consistent across the database. Do not edit this file. To change configuration parameters, use the **UPDATE DATABASE CONFIGURATION** and **RESET DATABASE CONFIGURATION** commands.
- History files.

  The DB2RHIST.ASC history file and its backup DB2RHIST.BAK contain history information about backups, restores, loading of tables, reorganization of tables, altering of a table space, and other changes to a database.

  The DB2TSCHG.HIS file contains a history of table space changes at a log-file level. For each log file, DB2TSCHG.HIS contains information that helps to identify which table spaces are affected by the log file. Table space recovery uses information from this file to determine which log files to process during table space recovery. You can examine the contents of history files in a text editor.
- Logging-related files.

  The global log control files, SQLOGCTL.GLFH.1, SQLOGCTL.GLFH.2, contain recovery information at the database level, for example, information related to the addition of new members while the database is offline and maintaining a common log chain across members. The log files themselves are stored in the LOGSTREAM*xxxx* directories (one for each member) in the partition-global directory.

- Locking files.

  The instance database lock files, SQLINSLK,and SQLTMPLK, help to ensure that a database is used by only one instance of the database manager.
- Automatic storage containers

## Member-specific directory

The member-specific directory has the path: /NODE*xxxx*/SQL*xxxx*/MEMBER*xxxx*

This directory contains objects associated with the first database created, and subsequent databases are given higher numbers: SQL00002, and so on. These subdirectories differentiate databases created in this instance on the directory that you specified in the **CREATE DATABASE** command.

The database directory contains the following files:

- Buffer pool information files.

  The files SQLBP.1 and SQLBP.2 contain buffer pool information. These files are duplicates of each other for backup purposes.
- Local event monitor files.
- Logging-related files.

  The log control files, SQLOGCTL.LFH.1, its mirror copy SQLOGCTL.LFH.2, and SQLOGMIR.LFH, contain information about the active logs. In a DB2 pureScale environment, each member has its own log stream and set of local LFH files, which are stored in each member-specific directory.

  **Tip:** Map the log subdirectory to disks that you are not using for your data. By doing so, you might restrict disk problems to your data or the logs, instead of having disk problems for both your data and the logs. Mapping the log subdirectory to disks that you are not using for your data can provide a substantial performance benefit because the log files and database containers do not compete for movement of the same disk heads. To change the location of the log subdirectory, use the **newlogpath** database configuration parameter.
- The local configuration file.

  The local SQLDBCONF file contains database configuration information. Do not edit this file. To change configuration parameters, use the **UPDATE DATABASE CONFIGURATION** and **RESET DATABASE CONFIGURATION** commands.

At the same time a database is created, a detailed deadlocks event monitor is also created. In an Enterprise Server Edition environment and in partitioned database environments, the detailed deadlocks event monitor files are stored in the database directory of the catalog node. In a DB2 pureScale environment, the detailed deadlocks event monitor files are stored in the partition-global directory. When the event monitor reaches its maximum number of files to output, it will deactivate and a message is written to the notification log. This prevents the event monitor from using too much disk space. Removing output files that are no longer needed allows the event monitor to activate again on the next database activation.

## Additional information for SMS database directories in non-automatic storage databases

In non-automatic storage databases, the SQLT* subdirectories contain the default System Managed Space (SMS) table spaces:

- SQLT0000.0 subdirectory contains the catalog table space with the system catalog tables.
- SQLT0001.0 subdirectory contains the default temporary table space.
- SQLT0002.0 subdirectory contains the default user data table space.

Each subdirectory or container has a file created in it called SQLTAG.NAM. This file marks the subdirectory as being in use so that subsequent table space creation does not attempt to use these subdirectories.

In addition, a file called SQL*.DAT stores information about each table that the subdirectory or container contains. The asterisk (*) is replaced by a unique set of digits that identifies each table. For each SQL*.DAT file there might be one or more of the following files, depending on the table type, the reorganization status of the table, or whether indexes, LOB, or LONG fields exist for the table:

- SQL*.BKM (contains block allocation information if it is an MDC or ITC table)
- SQL*.LF (contains LONG VARCHAR or LONG VARGRAPHIC data)
- SQL*.LB (contains BLOB, CLOB, or DBCLOB data)
- SQL*.XDA (contains XML data)
- SQL*.LBA (contains allocation and free space information about SQL*.LB files)
- SQL*.INX (contains index table data)
- SQL*.IN1 (contains index table data)
- SQL*.DTR (contains temporary data for a reorganization of an SQL*.DAT file)
- SQL*.LFR (contains temporary data for a reorganization of an SQL*.LF file)
- SQL*.RLB (contains temporary data for a reorganization of an SQL*.LB file)
- SQL*.RBA (contains temporary data for a reorganization of an SQL*.LBA file)

## Node directory

The database manager creates the *node directory* when the first database partition is cataloged.

To catalog a database partition, use the **CATALOG NODE** command.

**Note:** In a DB2 pureScale environment, it is not necessary to run the CATALOG NODE command, because the DB2 pureScale Feature acts as a single partition.

To list the contents of the local node directory, use the **LIST NODE DIRECTORY** command.

The node directory is created and maintained on each database client. The directory contains an entry for each remote database partition server having one or more databases that the client can access. The DB2 client uses the communication end point information in the node directory whenever a database connection or instance attachment is requested.

The entries in the directory also contain information about the type of communication protocol to be used to communicate from the client to the remote database partition. Cataloging a local database partition creates an alias for an instance that resides on the same computer.

## Local database directory

A *local database directory* file exists in each path (or "drive" for Windows operating systems) in which a database has been defined. This directory contains one entry for each database accessible from that location.

Each entry contains:

- The database name provided with the **CREATE DATABASE** command
- The database alias name (which is the same as the database name, if an alias name is not specified)
- A comment describing the database, as provided with the **CREATE DATABASE** command
- The name of the root directory for the database
- Other system information.

## Directory structure for your installed DB2 database product (Windows)

When you install DB2 database products, you can specify a DB2 database product installation path or else use the default path. After installation, DB2 objects are created in these directories.

Follow these steps to verify the DB2 product you have installed on Windows.

1. From a command prompt, type the **regedit** command. The Registry Editor window opens.
2. Expand **HKEY_LOCAL_MACHINE** > **Software** > **IBM** > **DB2**

The DB2 product you have installed will be displayed.

The following table shows the location of DB2 objects after a default installation.

*Table 5. DB2 objects and their locations*

| DB2 Object | Location |
|---|---|
| DAS information | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1` |
| Database configuration file `SQLDBCON` | `C:\DB2\NODE0000\SQL00001` |
| Database directory<br><br>Contains files needed for:<br>• buffer pool information<br>• history information<br>• log control files<br>• storage path information<br>• table space information | `C:\DB2\NODE0000\SQL00001` |

*Table 5. DB2 objects and their locations (continued)*

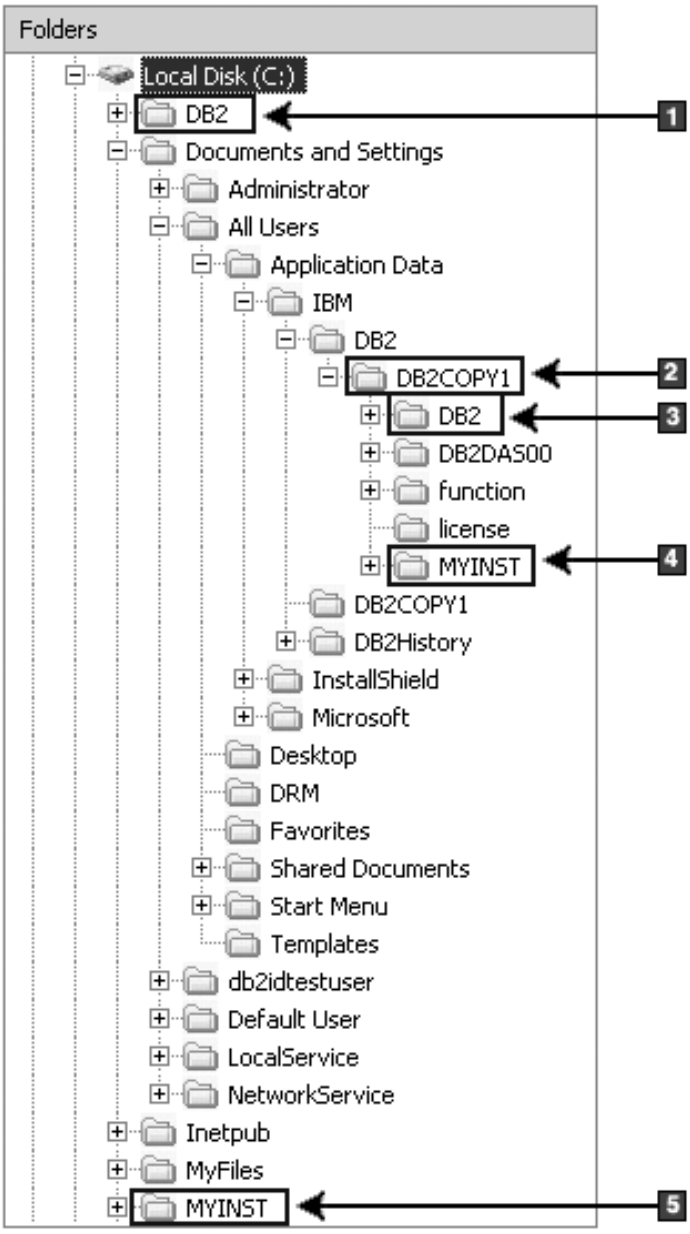
| DB2 Object | Location |
|---|---|
| Database manager configuration file `db2systm` | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2` |
| DB2 commands | `C:\Program Files\IBM\SQLLIB\BIN` |
| DB2 error messages file db2diag log files | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2` |
| DB2 installation path | `C:\Program Files\IBM\SQLLIB` |
| Directory for event monitor data | `C:\DB2\NODE0000\SQL00001\DB2EVENT` |
| Directory for transaction log files | `C:\DB2\NODE0000\SQL00001\LOGSTREAM0000` |
| Installation log file | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\Administrator\My Documents\DB2LOG`<br>• For Windows Vista and later operating systems: `C:\Users\`*`USER_NAME`*`\Documents\DB2LOG` |
| Instance | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2` |
| Instance information | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2` |
| Node directory | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2\SQLNODIR`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2\SQLNODIR` |
| Local database directory for the instance called DB2 | `C:\DB2\NODE0000\SQLDBDIR` |

*Table 5. DB2 objects and their locations (continued)*

| DB2 Object | Location |
| --- | --- |
| Partitioned database environment file `db2nodes.cfg` | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2` |
| System database directory | • For Windows XP and Windows 2003 operating systems: `C:\Documents and Settings\All Users\Application Data\IBM\DB2\DB2COPY1\DB2\SQLDBDIR`<br>• For Windows Vista and later operating systems: `C:\ProgramData\IBM\DB2\DB2COPY1\DB2\SQLDBDIR` |

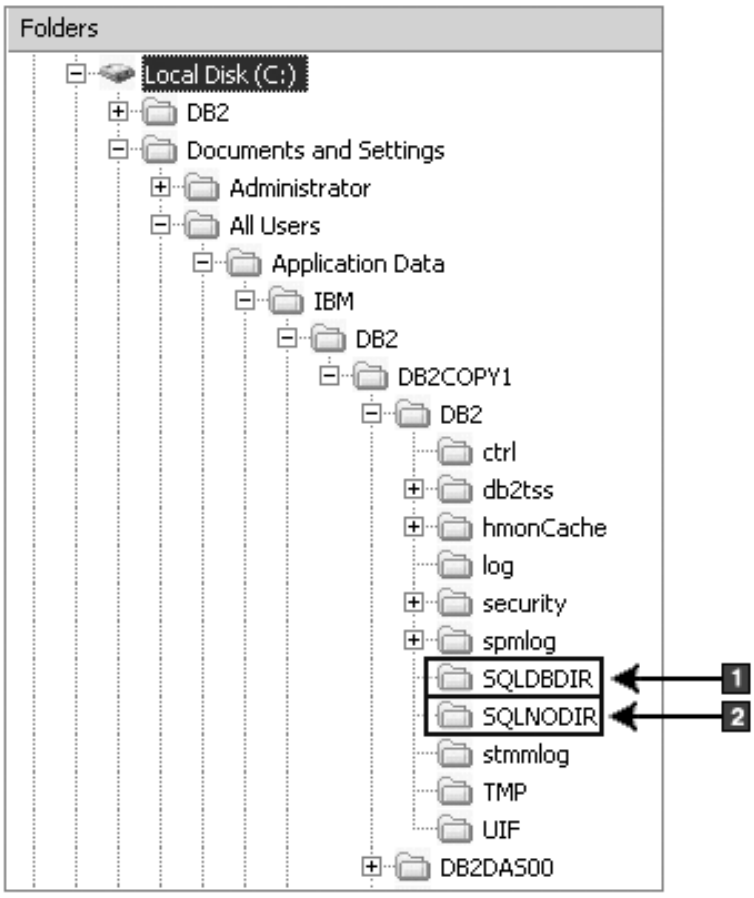
The following figures illustrate an example of the DB2 directory structure after installation on Windows XP or Windows 2003 operating systems using the default options. In these figures, there are two instances, DB2 and MYINST. The directories `DB2` and `MYINST` under the local disk `C:` will only appear if a database has been created under the appropriate instance.

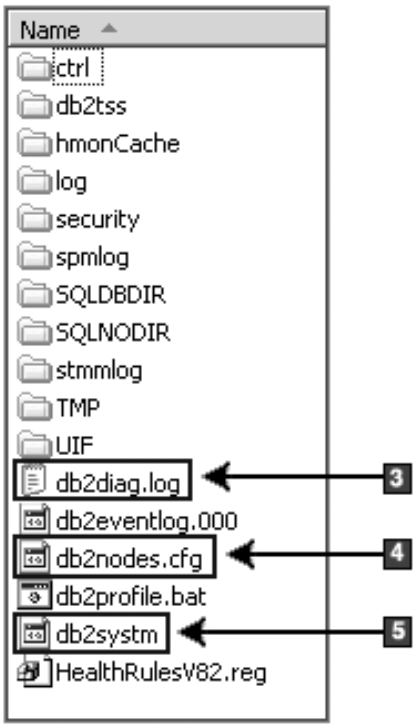
## Directory structure - instance information




1. Contains the databases created under the `C:` drive for the instance named DB2.
2. Contains the information for the DAS.
3. Contains the instance information for the instance named DB2.
4. Contains the instance information for the instance named MYINST.
5. Contains the databases created under the `C:` drive for the instance named MYINST.

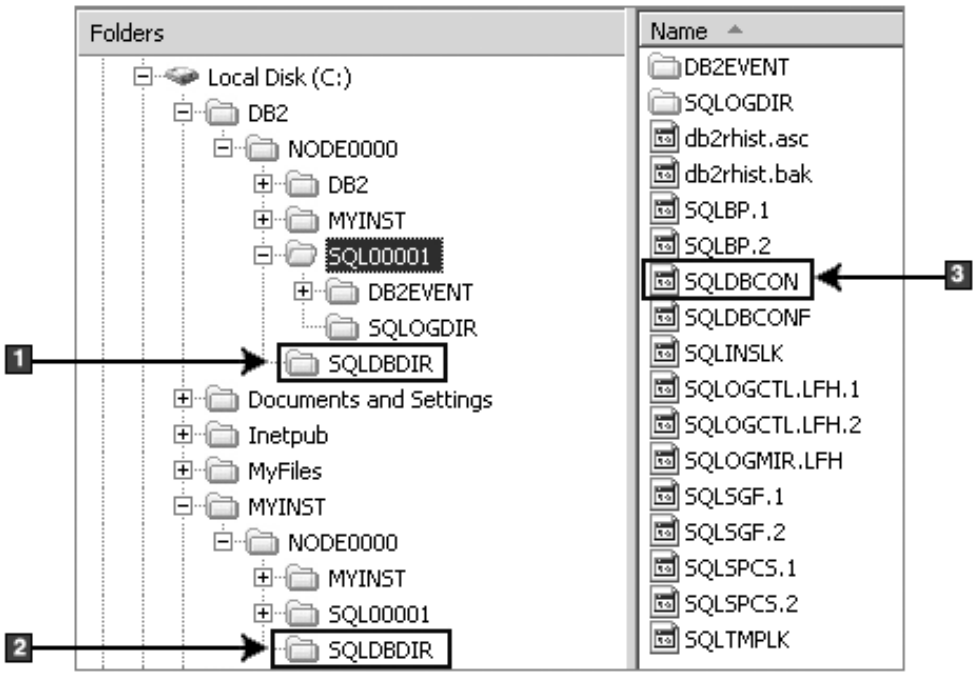
## Directory structure - directory information





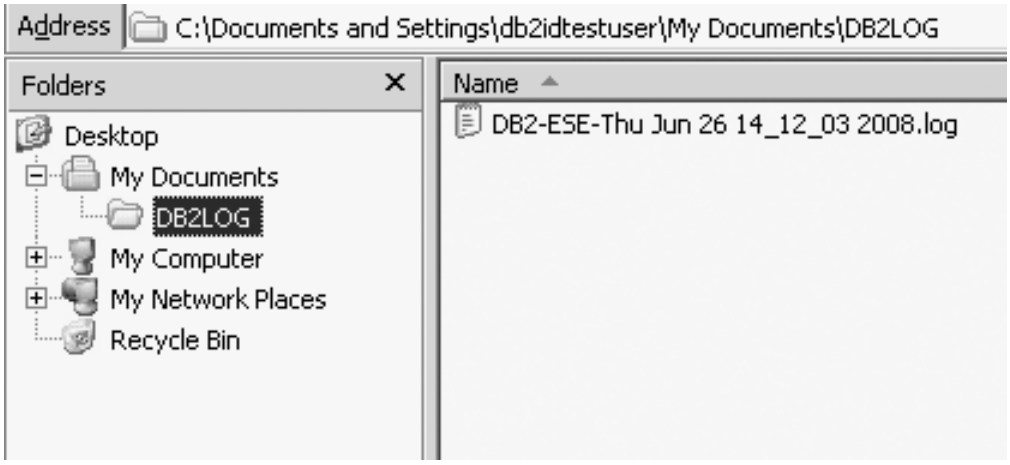

1. System database directory
2. Node directory
3. The db2diag log files DB2 error messages.
4. The `db2nodes.cfg` file is used in a partitioned database environment.
5. Database manager configuration file

### Directory structure - local directory information




1. Local database directory for the instance DB2
2. Local database directory for the instance MYINST
3. Database configuration file

### Directory structure - installation log file location




## Directory structure for your installed DB2 database product (Linux)

During a root installation, you can specify where the subdirectories and files for the DB2 database product will be created. For non-root installations, you cannot choose where DB2 products are installed; you must use the default locations.

**Note:** For non-root installations, all DB2 files (program files and instance files) are located in or beneath the *$HOME*/sqllib directory, where *$HOME* represents the non-root user's home directory.

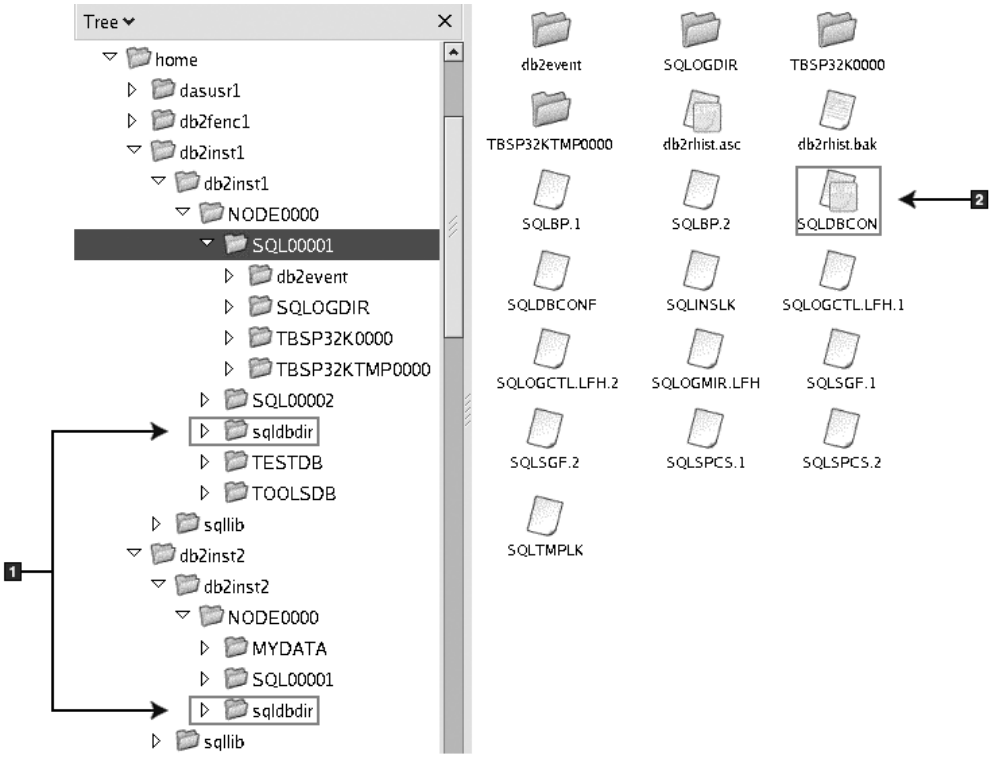
After installation, the DB2 objects are created in various directories. The following table shows the location of DB2 objects after a default root installation.

*Table 6. Location of DB2 objects after a default root installation*

| DB2 Object | Location |
|---|---|
| DAS home directory | home/dasusr1 |
| DAS information | home/dasusr1/das |
| Database configuration file SQLDBCON | home/db2inst1/db2inst1/NODE0000/SQL00001 |
| Database directory<br><br>Contains files needed for:<br>• buffer pool information<br>• history information<br>• log control files<br>• storage path information<br>• table space information | home/db2inst1/db2inst1/NODE0000/SQL00001 |
| Database manager configuration file db2systm | home/db2inst1/sqllib |
| DB2 commands | /opt/IBM/db2/V10.1/bin |
| DB2 error messages file (db2diag log file) | home/db2inst1/sqllib/db2dump |
| DB2 installation path | default is /opt/IBM/db2/V10.1 |
| Directory for event monitor data | home/db2inst1/db2inst1/NODE0000/<br>SQL00001/db2event |
| Directory for transaction log files | home/db2inst1/db2inst1/NODE0000/<br>SQL00001/LOGSTREAM0000 |
| Installation log file db2install.history | /opt/IBM/db2/V10.1/install/logs |
| Instance home directory | home/db2inst1 |
| Instance information | home/db2inst1/sqllib |
| Local database directory for the instance | home/db2inst1/db2inst1/NODE0000/sqldbdir |
| Partitioned database environment file db2nodes.cfg | home/db2inst1/sqllib |
| System database directory | home/db2inst1/sqllib/sqldbdir |

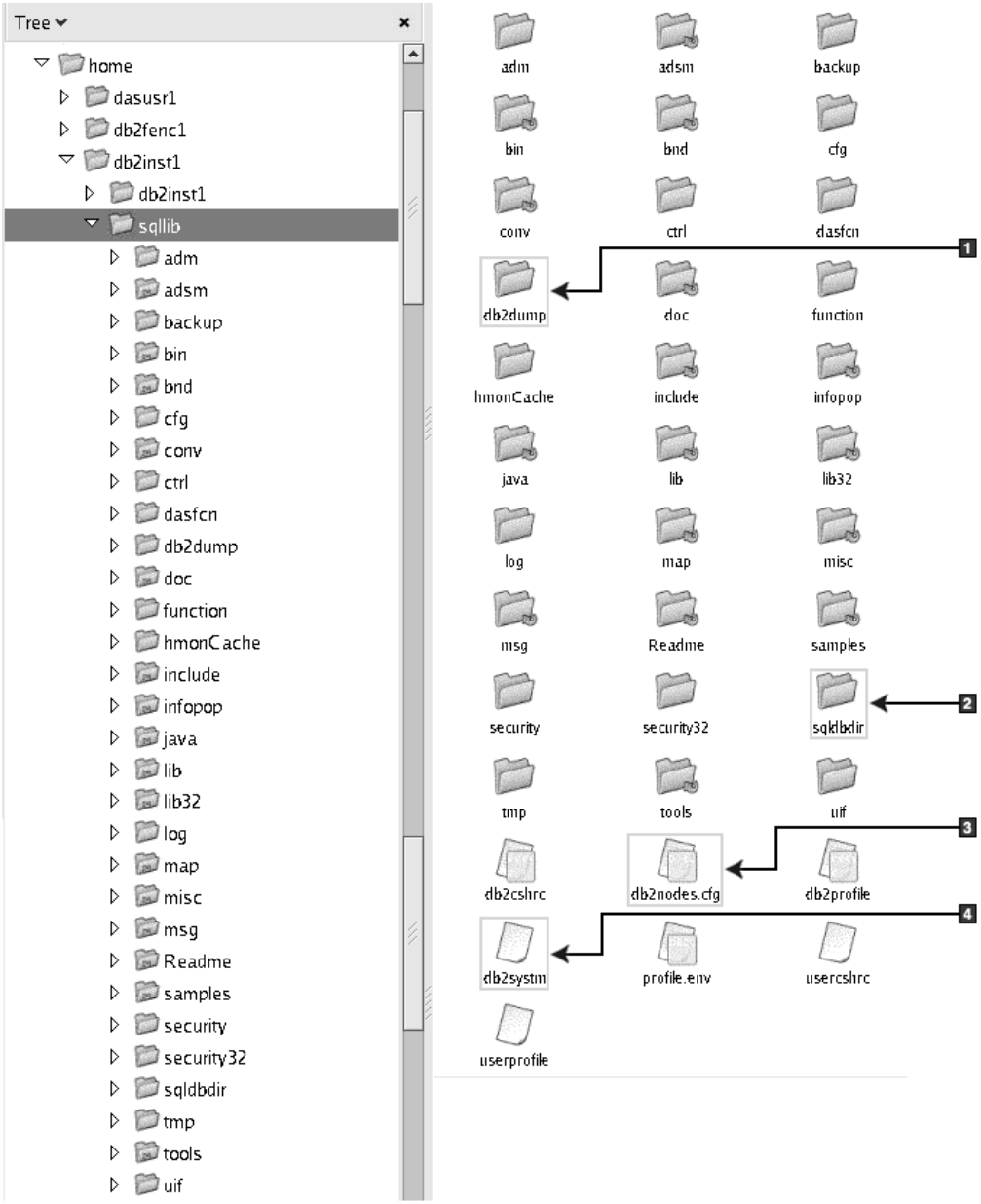
The following figures illustrate an example of the DB2 directory structure after a root installation. In these examples, there are two instances, db2inst1 and db2inst2.

## Directory structure - default local database directory information for the DB2 instance db2inst1




1. Local database directories.
2. Database configuration file

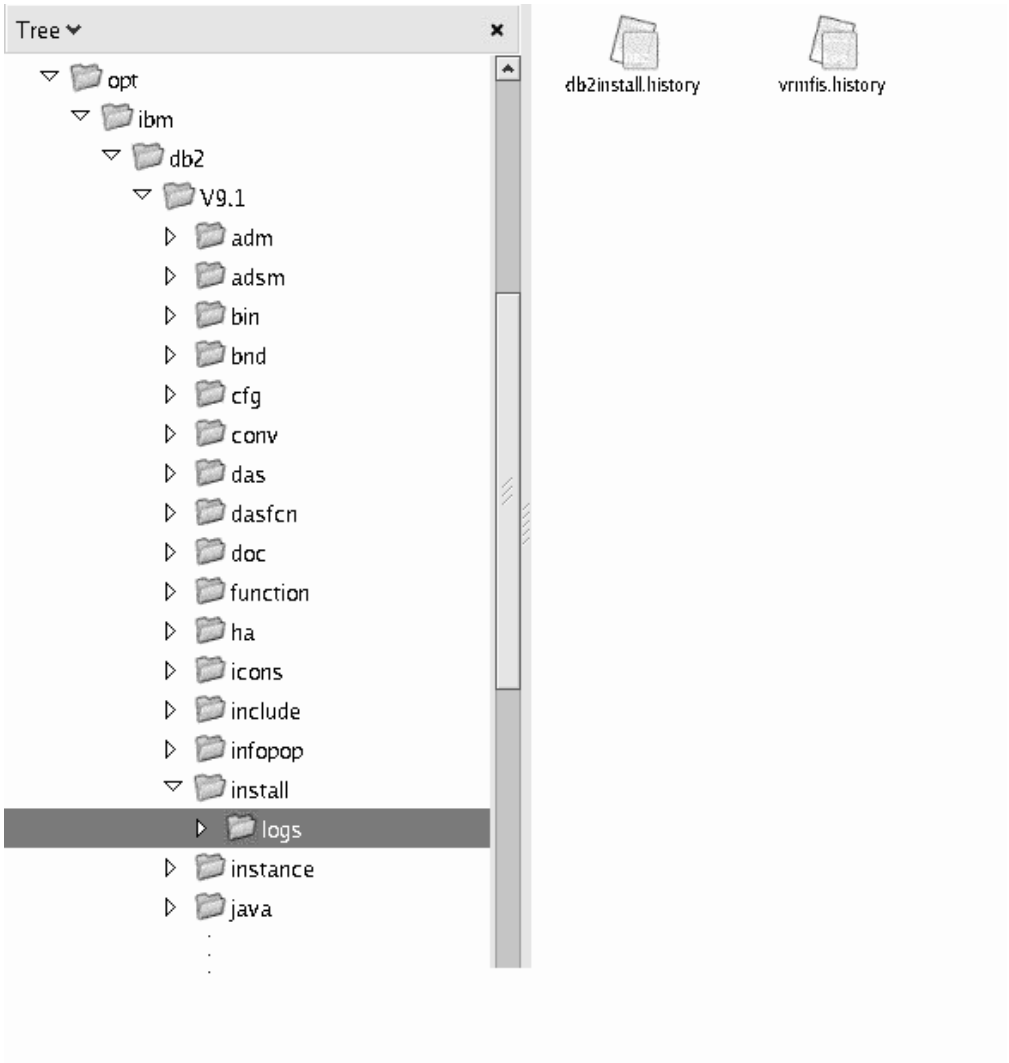
## Directory structure - directory information for the DB2 instance db2inst1




1. The db2diag log files in this directory logs DB2 error messages.
2. System database directory
3. The `db2nodes.cfg` file is used in a partitioned database environment.
4. Database manager configuration file

The `dasusr1` directory contains the DB2 administration server (DAS) configuration files and the symbolic links to DB2 core files.

### Directory structure - install log file location




This figure illustrates the location of your install log file. If there were multiple installations under the same installation path, the `db2install.history` file will be indexed as `db2install.history.n` where *n* represents a four digit number, for example, `0000`, or `0001`.

The DB2 installation directory `/opt/IBM/db2/V10.1` contains the installed DB2 files.

## System database directory

A *system database directory* file exists for each instance of the database manager, and contains one entry for each database that has been cataloged for this instance.

Databases are implicitly cataloged when the **CREATE DATABASE** command is issued and can also be explicitly cataloged with the **CATALOG DATABASE** command.

For each database created, an entry is added to the directory containing the following information:

- The database name provided with the **CREATE DATABASE** command

- The database alias name (which is the same as the database name, if an alias name is not specified)
- The database comment provided with the **CREATE DATABASE** command
- The location of the `local database directory`
- An indicator that the database is *indirect*, which means that it resides on the current database manager instance
- Other system information.

On Linux and UNIX platforms and in a partitioned database environment, you must ensure that all database partitions always access the same system database directory file, `sqldbdir`, in the `sqldbdir` subdirectory of the home directory for the instance. Unpredictable errors can occur if either the system database directory or the system intention file `sqldbins` in the same `sqldbdir` subdirectory are symbolic links to another file that is on a shared file system.

## Creating databases

You create a database using the **CREATE DATABASE** command. To create a database from a client application, call the sqlecrea API. All databases are created with the default storage group IBMSTOGROUP, unless you specify otherwise. Automatic storage managed table spaces use storage groups for their storage definitions.

### Before you begin

The DB2 database manager must be running. Use the **db2start** command to start the database manager.

It is important to plan your database, keeping in mind the contents, layout, potential growth, and how it will be used before you create it. After it has been created and populated with data, changes can be made.

The following database privileges are automatically granted to PUBLIC: CREATETAB, BINDADD, CONNECT, IMPLICIT_SCHEMA, and SELECT on the system catalog views. However, if the **RESTRICTIVE** option is present, no privileges are automatically granted to PUBLIC. For more information about the **RESTRICTIVE** option, see the **CREATE DATABASE** command.

Restrictions

- Storage paths cannot be specified using relative path names; you must use absolute path names. The storage path can be up to 175 characters long.
- On Windows operating systems, the database path must be a drive letter only, unless the **DB2_CREATE_DB_ON_PATH** registry variable is set to `YES`.
- If you do not specify a database path using the **DBPATH ON** clause of the **CREATE DATABASE** command, the database manager uses the first storage path specified for the **ON** clause for the database path. (On Windows operating systems, if this clause is specified as a path, and if the **DB2_CREATE_DB_ON_PATH** registry variable is not set to `YES`, you receive a SQL1052N error message.) If no **ON** clause is specified, the database is created on the default database path that is specified in the database manager configuration file (**dftdbpath** parameter). The path is also used as the location for the single storage path associated with the database.
- For partitioned databases, you must use the same set of storage paths on each database partition (unless you use database partition expressions).

- Database partition expressions are not valid in database paths, whether you specify them explicitly by using the **DBPATH ON** clause of the **CREATE DATABASE** command, or implicitly by using a database partition expression in the first storage path.
- A storage group must have at least one storage path associated with it.

**Note:** Although, you can create a database specifying the **AUTOMATIC STORAGE NO** clause, the **AUTOMATIC STORAGE** clause is deprecated and might be removed from a future release.

## About this task

When you create a database, each of the following tasks are done for you:

- Setting up of all the system catalog tables that are needed by the database
- Allocation of the database recovery log
- Creation of the database configuration file and the default values are set
- Binding of the database utilities to the database

## Procedure

- To create a database from a client application, call the sqlecrea API.
- To create a database using the command line processor, issue the **CREATE DATABASE** command.

  For example, the following command creates a database called PERSON1, in the default location, with the associated comment "Personnel DB for BSchiefer Co".

  ```
  CREATE DATABASE person1
    WITH "Personnel DB for BSchiefer Co"
  ```

- To create a database using IBM Data Studio, right-click the instance on which you want to create the database and select the task assistant to the create it. For more information, see IBM Data Studio: Administering databases with task assistants.

## Example

*Example 1: Creating a database on a UNIX or Linux operating system*:

To create a database named TESTDB1 on path `/DPATH1` using `/DATA1` and `/DATA2` as the storage paths defined to the default storage group IBMSTOGROUP, use the following command:

```
CREATE DATABASE TESTDB1 ON '/DATA1','/DATA2' DBPATH ON '/DPATH1'
```

*Example 2: Creating a database on a Windows operating system, specifying both storage and database paths*:

To create a database named TESTDB2 on drive `D:`, with storage on `E:\DATA`, use the following command:

```
CREATE DATABASE TESTDB2 ON 'E:\DATA' DBPATH ON 'D:'
```

In this example, `E:\DATA` is used as both the storage path defined to the default storage group IBMSTOGROUP and the database path.

*Example 3: Creating a database on a Windows operating system, specifying only a storage path*:

To create a database named TESTDB3 with storage on drive F:, use the following command:

```
CREATE DATABASE TESTDB3 ON 'F:'
```

In this example, F: is used as both the storage path defined to the default storage group IBMSTOGROUP and the database path.

If you specify a directory name such as F:\DATA for the storage path, the command fails, because:

1. When **DBPATH** is not specified, the storage path -- in this case, F:\DATA -- is used as the database path
2. On Windows, the database path can only be a drive letter (unless you change the default for the **DB2_CREATE_DB_ON_PATH** registry variable from NO to YES).

If you want to specify a directory as the storage path on Windows operating systems, you must also include the **DBPATH ON** drive clause, as shown in Example 2.

*Example 4: Creating a database on a UNIX or Linux operating system without specifying a database path*:

To create a database named TESTDB4 with storage on /DATA1 and /DATA2, use the following command:

```
CREATE DATABASE TESTDB4 ON '/DATA1','/DATA2'
```

In this example, /DATA1 and /DATA2 are used as the storage paths defined to the default storage group IBMSTOGROUP and /DATA1 is the database path.

## What to do next

**Configuration Advisor**

The Configuration Advisor helps you to tune performance and to balance memory requirements for a single database per instance by suggesting which configuration parameters to modify and providing suggested values for them. The Configuration Advisor is automatically invoked by default when you create a database.

You can override this default so that the configuration advisor is not automatically invoked by using one of the following methods:

- Issue the **CREATE DATABASE** command with the **AUTOCONFIGURE APPLY NONE** parameter.
- Set the **DB2_ENABLE_AUTOCONFIG_DEFAULT** registry variable to NO:

  ```
  db2set DB2_ENABLE_AUTOCONFIG_DEFAULT=NO
  ```

  However, if you specify the **AUTOCONFIGURE** parameter with the **CREATE DATABASE** command, the setting of this registry variable is ignored.

Also, the following automatic features are enabled by default when you create a database:

- Automatic storage
- Automatic background statistics collection
- Automatic real-time statistics collection
- Self-tuning memory (single-partition environments)

**Event Monitor**

At the same time a database is created, a detailed deadlocks event monitor

is also created. As with any monitor, there is extra processing time and resources associated with this event monitor. If you do not want the detailed deadlocks event monitor, then the event monitor can be dropped by using the command:

```
DROP EVENT MONITOR db2detaildeadlock
```

To limit the amount of disk space that this event monitor consumes, the event monitor deactivates, and a message is written to the administration notification log, once it has reached its maximum number of output files. Removing output files that are no longer needed allows the event monitor to activate again on the next database activation.

**Remote databases**

You can create a database in a different, possibly remote, instance. To create a database at another (remote) database partition server, you must first attach to that server. A database connection is temporarily established by the following command during processing:

```
CREATE DATABASE database_name AT DBPARTITIONNUM options
```

In this type of environment, you can perform instance-level administration against an instance other than your default instance, including remote instances. For instructions on how to do this, see the **db2iupdt** (update instance) command.

**Database code pages**

By default, databases are created in the UTF-8 (Unicode) code set.

To override the default code page for the database, it is necessary to specify the required code set and territory when creating the database. See the **CREATE DATABASE** command or the sqlecrea API for information about setting the code set and territory.

# Viewing the local or system database directory files

Use the **LIST DATABASE DIRECTORY** command to view the information associated with the databases that you have on your system.

## Before you begin

Before viewing either the local or system database directory files, you must have previously created an instance and a database.

## Procedure

- To see the contents of the local database directory file, issue the following command:

  ```
  LIST DATABASE DIRECTORY ON location
  ```

  where *location* specifies the location of the database.
- To see the contents of the system database directory file, issue the **LIST DATABASE DIRECTORY** command *without* specifying the location of the database directory file.

# Client-to-server connectivity

Configuring client-to-server communications for the IBM data server client and DB2 database server products requires and understanding of the components and type of connections.

## Components and scenarios

The basic components involved in client-to-server communications are described in the following section:

- **Client.** This refers to the initiator of the communications. This role can be filled by any of the following DB2 products or components:
  - IBM Data Server Driver Package
  - IBM Data Server Client or IBM Data Server Runtime Client.
  - DB2 Connect Personal Edition: This product is a superset of the IBM Data Server Client.
  - a DB2 server product: A DB2 server is a superset of the Data Server Client.
- **Server.** This refers to the receiver of the communications request from the client. This role is normally filled by a DB2 for Linux, UNIX, and Windows server product. When DB2 Connect products are present, the term *server* can also mean a DB2 server on a midrange or mainframe platform.
- **Communications protocol.** This refers to the protocol used to send data between the client and server. The DB2 product supports several protocols:
  - TCP/IP. A further distinction can be made between the version: TCP/IPv4 or TCP/IPv6.
  - Named Pipes. This option is available on Windows only.
  - IPC (interprocess communications). This protocol is used for local connections.

There are also some additional components encountered in some environments:

- **DB2 Connect gateway.** This refers to a DB2 Connect server product that provides a gateway by which IBM data server client can connect to DB2 servers on midrange and mainframe products.
- **LDAP (Lightweight Directory Access Protocol).** In an LDAP-enabled environment, it is not necessary to configure client-to-server communications. When a client attempts to connect to a database, if the database does not exist in the database directory on the local machine then the LDAP directory is searched for information required to connect to the database.

The following scenarios illustrate examples of situations covered by client-to-server communications:

- Data Server Client establishes communications with a DB2 server using TCP/IP.
- Data Server Runtime Client establishes communications with a DB2 server using Named Pipes on a Windows network.
- DB2 server establishes communications with another DB2 server via some communications protocol.
- Data Server Client establishes communications with a mainframe DB2 server via a DB2 Connect server using TCP/IP.

When setting up a server to work with development environments (such as IBM Data Studio), you might encounter error message SQL30081N at the initial DB2 connection. A possible root cause is that the firewall at the remote database server

has prevented the connection from being established. In this case, verify the firewall is properly configured to accept connection requests from the client.

## Types of connections

Generally speaking, references to setting up client-to-server communications refer to *remote connections*, rather than *local connections*.

A *local connection* is a connection between a database manager instance and a database managed by that instance. In other words, the CONNECT statement is issued from the database manager instance to itself. Local connections are distinctive because no communications setup is required and IPC (interprocess communications) is used.

A *remote connection* is one where the client issuing the CONNECT statement to a database is in a different location from the database server. Commonly, the client and server are on different machines. However, remote connections are possible within the same machine if the client and server are in different instances.

Another less common type of connection is a *loopback connection*. This is a type of remote connection where the connection is configured from a DB2 instance (the client) to the same DB2 instance (the server).

## Configuration of client-to-server communications

You can configure client-to-server communications by using the command line tools which consist of the Command Line Processor (CLP), the **db2cfexp** (configuration export) command, and the **db2cfimp** (configuration import) command.

Use the following table to identify the appropriate configuration method.

*Table 7. Tools and methods for configuring a client-to-server connection*

| Type of configuration task | CLP |
|---|---|
| Configure a client by entering information manually | Configure client-to-server connections by using the **CATALOG TCPIP/TCPIP4/TCPIP6 NODE command** and the **CATALOG DATABASE** command. |
| Use the connection settings for one client as the basis for configuring additional clients | 1. Create a client profile by issuing the **db2cfexp** command.<br>2. Configure database connections using a client profile by issuing the **db2cfimp** command. |

**Note:** Use *Profiles* to configure client-to-server communications. The types of profiles are:

- A *client profile* is a file that contains settings for a client. Settings can include:
  - Database connection information (including CLI or ODBC settings).
  - Client settings (including database manager configuration parameters and DB2 registry variables).
  - CLI or ODBC common parameters.
- A *server profile* is similar to a client profile but contains settings for a server.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for configuring automatic maintenance. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

## IBM data server client and driver types

There are several types of IBM data server clients and drivers available. Each provides a particular type of support.

The IBM data server client and driver types are as follows:

- IBM Data Server Driver Package
- IBM Data Server Driver for JDBC and SQLJ
- IBM Data Server Driver for ODBC and CLI
- IBM Data Server Runtime Client
- IBM Data Server Client

The DB2 Connect Personal Edition product includes all the functionality of IBM Data Server Client and connects to midrange and mainframe databases. DB2 Connect capability can be added to any client or driver and the recommended approach is to use the DS driver.

Each IBM data server client and driver provides a particular type of support:

- For Java applications only, use IBM Data Server Driver for JDBC and SQLJ.
- For applications using ODBC or CLI only, use IBM Data Server Driver for ODBC and CLI. (Also referred to as cli driver.)
- For applications using ODBC, CLI, .NET, OLE DB, PHP, Ruby, JDBC, or SQLJ, use IBM Data Server Driver Package.
- For applications using DB2CI, use IBM Data Server Client.
- If you need DB2 Command Line Processor Plus (CLPPlus) support, use IBM Data Server Driver Package.
- To have command line processor (CLP) support and basic client support for running and deploying applications, use IBM Data Server Runtime Client. Alternatively use CLPPlus,which is a component of the recommended IBM Data Server Driver Package.
- To have support for database administration, and application development using an application programming interface (API), such as ODBC, CLI, .NET, or JDBC, use IBM Data Server Client.

### IBM Data Server Driver Package

IBM Data Server Driver Package is a lightweight deployment solution that provides runtime support for applications using ODBC, CLI, .NET, OLE DB, PHP, Ruby, JDBC, or SQLJ, without the need to install the Data Server Runtime Client or Data Server Client. This driver has a small footprint and is designed to be redistributed by independent software vendors (ISVs), and to be used for application distribution in mass deployment scenarios typical of large enterprises.

The IBM Data Server Driver Packageinclude the following capabilities:

- DB2 Command Line Processor Plus (CLPPlus), for dynamically creating, editing, and running SQL statements and scripts.
- Support for applications that use ODBC, CLI, PHP, or Ruby to access databases.

- On Windows operating systems, IBM Data Server Driver Package also provides support for applications that use .NET or OLE DB to access databases. In addition, this driver package is available as an installable image. Merge modules are available to allow you to easily embed the driver in a Windows Installer-based installation.
- Support for client applications and applets that are written in the Java language using JDBC and for embedded SQL for Java (SQLJ).
- Support for running embedded SQL applications. No precompiler or bind capabilities are provided.
- Application header files to rebuild the PHP, Ruby, Python, and Perl drivers. The Python and Perl drivers are not available in IBM Data Server Driver Package; however, you can download and build these drivers by using the header files.
- Support for DB2 Interactive CLI through the **db2cli** command.
- Support for DRDA traces through the **db2drdat** command.
- This client package also supports IBM Informix servers.

## IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ is the default driver for Java stored procedures and user-defined functions. This driver provides support for client applications and applets that are written in Java using JDBC to access local or remote servers, and SQLJ for embedded static SQL in Java applications. This driver is a prerequisite for IBM InfoSphere Optim pureQuery Runtime, which provides static support for Java, enables optimized data access using the pureQuery API, and is supported by a full integrated development environment (IDE) for Java database application development using IBM InfoSphere Optim Development Studio. (Both Optim products are available separately.)

## IBM Data Server Driver for ODBC and CLI

Data Server Driver for ODBC and CLI is a lightweight deployment solution designed for independent software vendors (ISV) deployments. This driver, also referred to as cli driver, provides runtime support for applications using ODBC API, or CLI API without need of installing the Data Server Client or the Data Server Runtime Client. This driver is available only as a tar file, not as an installable image. Messages are reported only in English.

## IBM Data Server Runtime Client

The IBM Data Server Runtime Client provides a way to run applications on remote databases. GUI tools are not shipped with the IBM Data Server Runtime Client.

Capabilities include the following ones:

- All the functionality from IBM Data Server Driver.
- The DB2 command line processor (CLP) for issuing commands. The CLP also provides a basic way to remotely administer servers.
- The ASNCLP command-line program to set up and administer all replication programs for Q replication and SQL replication.
- Support for common network communication protocols: TCP/IP, and Named Pipe.
- Smaller deployment footprint compared to that of the full IBM Data Server Client in terms of installation image size and disk space required.
- A catalog that stores information for connecting to databases and servers.

### IBM Data Server Client

IBM Data Server Client includes all the functionality of IBM Data Server Runtime Client, plus functionality for database administration, application development, and client/server configuration.

Capabilities include the following ones:

- The ability to prune the IBM Data Server Client image to reduce the installation image size on the Windows operating system.
- Replication tools to set up and administer all replication programs for Q replication and SQL replication. These tools are the Replication Center, the ASNCLP command-line program, and the Replication Alert Monitor tool. The Replication Center is available only on Linux and Windows operating systems.
- First Steps documentation for new users.
- Visual Studio tools.
- Application header files.
- Precompilers for various programming languages.
- Bind support.
- Samples and tutorials.

## Cataloging a database

This task describes how to catalog a database from a client by using the command line processor (CLP).

### Before you begin

Before a client application can access a remote database, the database must be cataloged on the client. When you create a database, the database is automatically cataloged on the server with a database alias that is the same as the database name, unless a different database alias was specified.

The information in the database directory, along with the information in the node directory (unless you are cataloging a local database where a node is not needed), is used on the IBM data server client to establish a connection to the remote database.

- You require a valid DB2 user ID. DB2 does not support using root authority to catalog a database.
- You must have System Administrative (SYSADM) or System Controller (SYSCTRL) authority, or have the **catalog_noauth** option set to ON.
- You need the following information when cataloging a *remote* database:
  - Database name
  - Database alias
  - Node name
  - Authentication type (optional)
  - Comment (optional)

  Refer to the parameter values worksheet for cataloging a database for more information about these parameters and to record the values that you use.
- The following parameter values are applicable when cataloging a *local* database:
  - Database name
  - Drive

– Database alias
– Authentication type (optional)
– Comment (optional)

Local databases can be uncataloged and recataloged at any time.

### Procedure

To catalog a database on the client:

1. Log on to the system with a valid DB2 user ID.
2. If you are using the DB2 database on a Linux or UNIX platform, set up the instance environment. Run the startup script:

   **For bash, Bourne or Korn shell**

   ```
   . INSTHOME/sqllib/db2profile
   ```

   **For C shell**

   ```
   source INSTHOME/sqllib/db2cshrc
   ```

   where: *INSTHOME* represents the home directory of the instance.
3. Start the DB2 command line processor. On Windows operating systems, issue the **db2cmd** command from a command prompt. On Linux or UNIX, issue the **db2** command from a command prompt.
4. Catalog the database by entering the following commands in the command line processor:

   ```
   db2 => catalog database database_name as database_alias at
      node node_name [ authentication auth_value ]
   ```

   where:
   - *database_name* represents the name of the database you want to catalog.
   - *database_alias* represents a local nickname for the database you want to catalog.
   - *node_name* represents a nickname you can set for the computer that has the database you want to catalog.
   - *auth_value* specifies the type of authentication that takes place when connecting to the database. This parameter defaults to the authentication type specified on the server. Specifying an authentication type can result in a performance benefit. Examples of valid values include: `SERVER`, `CLIENT`, `SERVER_ENCRYPT`, `KERBEROS`, `DATA_ENCRYPT`, `GSSPLUGIN` and `SERVER_ENCRYPT_AES`.

### Example

To catalog a remote database called SAMPLE so that it has the local database alias MYSAMPLE, on the node DB2NODE using authentication `SERVER`, enter the following commands:

```
db2 => catalog database sample as mysample at node db2node
       authentication server
db2 => terminate
```

## Connecting to a database

### Before you begin

After cataloging the node and the database, connect to the database to test the connection. Before testing the connection:

- The database node and database must be cataloged.
- The values for *userid* and *password* must be valid for the system on which they are authenticated. The authentication parameter on the client is be set to match the value on the server or it can be left unspecified. If an authentication parameter is not specified, the client will default to SERVER_ENCRYPT. If the server does not accept SERVER_ENCRYPT, then the client retries using the value returned from the server. If the client specifies an authentication parameter value that doesn't match what is configured on the server, you will receive an error.
- The database manager must be started with the correct protocol defined in the **DB2COMM** registry variable. If it is not started, then you can start the database manager by entering the **db2start** command on the database server.

## Procedure

To test the client to server connection:

1. If you are using a Linux or UNIX platform, set up the instance environment. Run the startup script:

   **For bash, Bourne or Korn shell**

   ```
   . INSTHOME/sqllib/db2profile
   ```

   **For C shell**

   ```
   source INSTHOME/sqllib/db2cshrc
   ```

   where: *INSTHOME* represents the home directory of the instance.
2. Start the DB2 command line processor. On Windows, issue the **db2cmd** command from a command prompt. On Linux or UNIX, issue the **db2** command from a command prompt.
3. Type the following command on the client to connect to the remote database:

   ```
   db2 => connect to database_alias user userid
   ```

   For example, enter the following command:

   ```
   connect to mysample user jtris
   ```

   You will be prompted to enter your password.

## Example

If the connection is successful, you receive a message showing the name of the database to which you have connected. A message similar to the following is given:

```
Database Connection Information
Database server = DB2 9.1.0
SQL authorization ID = JTRIS
Local database alias = mysample
```

You can now work with the database. For example, to retrieve a list of all the table names listed in the system catalog table, enter the following SQL statement:

```
select tabname from syscat.tables
```

## What to do next

When you are finished using the database connection, enter the **connect reset** command to end the database connection.

# Chapter 17. Table spaces

A *table space* is a storage structure containing tables, indexes, large objects, and long data. They are used to organize data in a database into logical storage groupings that relate to where data is stored on a system. Table spaces are stored in database partition groups.

Using table spaces to organize storage offers a number of benefits:

**Recoverability**
Putting objects that must be backed up or restored together into the same table space makes backup and restore operations more convenient, since you can backup or restore all the objects in table spaces with a single command. If you have partitioned tables and indexes that are distributed across table spaces, you can backup or restore only the data and index partitions that reside in a given table space.

**More tables**
There are limits to the number of tables that can be stored in any one table space; if you have a need for more tables than can be contained in a table space, you need only to create additional table spaces for them.

**Automatic storage management**
With automatic storage table spaces table spaces, storage is managed automatically. The database manager creates and extends containers as needed.

**Ability to isolate data in buffer pools for improved performance or memory utilization**
If you have a set of objects (for example, tables, indexes) that are queried frequently, you can assign the table space in which they reside a buffer pool with a single CREATE or ALTER TABLESPACE statement. You can assign temporary table spaces to their own buffer pool to increase the performance of activities such as sorts or joins. In some cases, it might make sense to define smaller buffer pools for seldom-accessed data, or for applications that require very random access into a very large table; in such cases, data need not be kept in the buffer pool for longer than a single query

Table spaces consist of one or more *containers*. A container can be a directory name, a device name, or a file name. A single table space can have several containers. It is possible for multiple containers (from one or more table spaces) to be created on the same physical storage device (although you will get the best performance if each container you create uses a different storage device). If you are using automatic storage table spaces, the creation and management of containers is handled automatically by the database manager. If you are not using automatic storage table spaces, you must define and manage containers yourself.

Figure 5 on page 98 illustrates the relationship between tables and table spaces within a database, and the containers associated with that database.

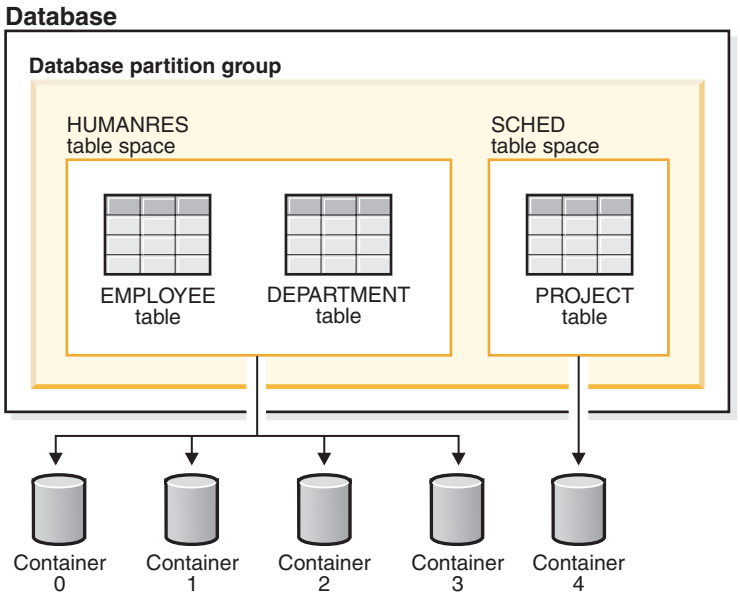



*Figure 5. Table spaces and tables in a database*

The EMPLOYEE and DEPARTMENT tables are in the HUMANRES table space, which spans containers 0, 1, 2 and 3. The PROJECT table is in the SCHED table space in container 4. This example shows each container existing on a separate disk.

The database manager attempts to balance the data load across containers. As a result, all containers are used to store data. The number of pages that the database manager writes to a container before using a different container is called the *extent size*. The database manager does not always start storing table data in the first container.

Figure 6 on page 99 shows the HUMANRES table space with an extent size of two 4 KB pages, and four containers, each with a small number of allocated extents. The DEPARTMENT and EMPLOYEE tables both have seven pages, and span all four containers.

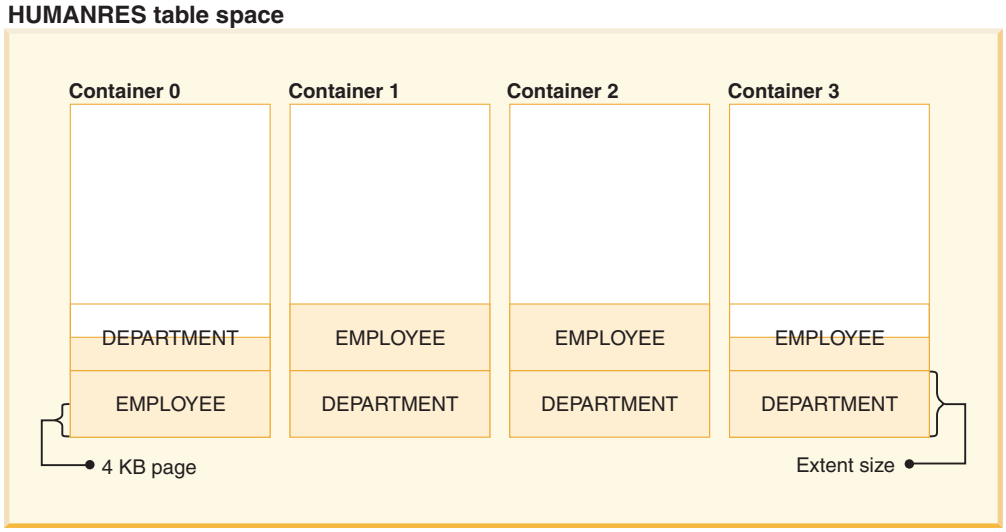



*Figure 6. Containers and extents in a table space*

## Table spaces for system, user and temporary data

Each database must have a minimal set of table spaces that are used for storing system, user and temporary data.

A database must contain at least three table spaces:

- A *catalog table space*
- One or more *user table spaces*
- One or more *temporary table spaces.*

### Catalog table spaces

A catalog table space contains all of the system catalog tables for the database. This table space is called SYSCATSPACE, and it cannot be dropped.

### User table spaces

A user table space contains user-defined tables. By default, one user table space, USERSPACE1, is created.

If you do not specify a table space for a table at the time you create it, the database manager will choose one for you. Refer to the documentation for the IN *tablespace-name* clause of the CREATE TABLE statement for more information.

The page size of a table space determines the maximum row length or number of columns that you can have in a table. The documentation for the CREATE TABLE statement shows the relationship between page size, and the maximum row size and column count. Before Version 9.1, the default page size was 4 KB. In Version 9.1 and following, the default page size can be one of the other supported values. The default page size is declared when creating a new database. Once the default page size has been declared, you are still free to create a table space with one page size for the table, and a different table space with a different page size for long or LOB data. If the number of columns or the row size exceeds the limits for a table

space's page size, an error is returned (SQLSTATE 42997).

### Temporary table spaces

A temporary table space contains temporary tables. Temporary table spaces can be *system temporary table spaces* or *user temporary table spaces*.

System temporary table spaces hold temporary data required by the database manager while performing operations such as sorts or joins. These types of operations require extra space to process the results set. A database must have at least one system temporary table space; by default, one system temporary table space called TEMPSPACE1 is created at database creation.

When processing queries, the database manager might need access to a system temporary table space with a page size large enough to manipulate data related to your query. For example, if your query returns data with rows that are 8KB long, and there are no system temporary table spaces with page sizes of at least 8KB, the query might fail. You might need to create a system temporary table space with a larger page size. Defining a temporary table space with a page size equal to that of the largest page size of your user table spaces will help you avoid these kinds of problems.

User temporary table spaces hold temporary data from tables created with a DECLARE GLOBAL TEMPORARY TABLE or CREATE GLOBAL TEMPORARY TABLE statement. They are not created by default at the time of database creation. They also hold instantiated versions of created temporary tables. To allow the definition of declared or created temporary tables, at least one user temporary table space should be created with the appropriate USE privileges. USE privileges are granted using the GRANT statement.

If a database uses more than one temporary table space and a new temporary object is needed, the optimizer will choose an appropriate page size for this object. That object will then be allocated to the temporary table space with the corresponding page size. If there is more than one temporary table space with that page size, then the table space will be chosen in a round-robin fashion, starting with one table space with that page size, and then proceeding to the next for the next object to be allocated, and so, returning to the first table space after all suitable table spaces have been used. In most circumstances, though, it is not recommended to have more than one temporary table space with the same page size.

## Types of table spaces

Table spaces can be set up in different ways depeding on the how you choose to manage their storage.

The three types of table spaces are known as:

- System managed space (SMS), in which the operating system's file manager controls the storage space once you have defined the location for storing database files
- Database managed space (DMS), in which the database manager controls the usage of storage space one you have allocated storage containers.
- Automatic storage table spaces, in which the database manager controls the creation of containers as needed.

Each can be used together in any combination within a database

## Automatic storage table spaces

With automatic storage table spaces, storage is managed automatically. The database manager creates and extends containers as needed.

**Note:** Although you can create a database specifying the AUTOMATIC STORAGE NO clause, the AUTOMATIC STORAGE clause is deprecated and might be removed from a future release.

Any table spaces that you create are managed as automatic storage table spaces unless you specify otherwise or the database was created using the AUTOMATIC STORAGE NO clause. With automatic storage table spaces, you are not required to provide container definitions; the database manager looks after creating and extending containers to make use of the storage allocated to the database. If you add storage to a storage group, new containers are automatically created when the existing containers reach their maximum capacity. If you want to make use of the newly-added storage immediately, you can rebalance the table space, reallocating the data across the new, expanded set of containers and stripe sets. Or, if you are less concerned about I/O parallelism, and just want to add capacity to your table space, you can forego rebalancing; in this case, as new storage is required, new stripe sets will be created.

Automatic storage table spaces can be created in a database using the CREATE TABLESPACE statement. By default, new tables spaces in a database are automatic storage table spaces, so the MANAGED BY AUTOMATIC STORAGE clause is optional. You can also specify options when creating the automatic storage table space, such as its initial size, the amount that the table space size will be increased when the table space is full, the maximum size that the table space can grow to, and the storage group it uses. Following are some examples of statements that create automatic storage table spaces:

```
CREATE TABLESPACE TS1
CREATE TABLESPACE TS2 MANAGED BY AUTOMATIC STORAGE
CREATE TEMPORARY TABLESPACE TEMPTS
CREATE USER TEMPORARY TABLESPACE USRTMP MANAGED BY AUTOMATIC STORAGE
CREATE LARGE TABLESPACE LONGTS
CREATE TABLESPACE TS3 INITIALSIZE 8K INCREASESIZE 20 PERCENT MANAGED BY AUTOMATIC STORAGE
CREATE TABLESPACE TS4 MAXSIZE 2G
CREATE TABLESPACE TS5 USING STOGROUP SG_HOT
```

Each of these examples assumes that the database for which these table spaces are being created has one or more defined storage groups. When you create a table space in a database that has no storage groups defined, you cannot use the MANAGED BY AUTOMATIC STORAGE clause; you must create a storage group, then try again to create your automatic storage table space.

## Comparison of automatic storage, SMS, and DMS table spaces

Automatic storage, SMS, and DMS table spaces offer different capabilities that can be advantageous in different circumstances.

**Important:** The SMS table space type has been deprecated in Version 10.1 for user-defined permanent table spaces and might be removed in a future release. The SMS table space type is not deprecated for catalog and temporary table spaces. For more information, see “SMS permanent table spaces have been deprecated” in *What’s New for DB2 Version 10.1*

**Important:** Starting with Version 10.1 Fix Pack 1, the DMS table space type is deprecated for user-defined permanent table spaces and might be removed in a future release. The DMS table space type is not deprecated for catalog and temporary table spaces. For more information, see "DMS permanent table spaces have been deprecated" in *What's New for DB2 Version 10.1*.

*Table 8. Comparison of SMS, DMS and automatic storage table spaces*

| | **Automatic storage table spaces** | **SMS table spaces** | **DMS table spaces** |
|---|---|---|---|
| How they are created | Created using the MANAGED BY AUTOMATIC STORAGE clause of the CREATE TABLESPACE statement, or by omitting the MANAGED BY clause entirely. If the automatic storage was enabled when the database was created, the default for any table space you create is to create it as an automatic storage table space unless you specify otherwise. | Created using the MANAGED BY SYSTEM clause of the CREATE TABLESPACE statement | Created using the MANAGED BY DATABASE clause of the CREATE TABLESPACE statement |
| Initial container definition and location | You do not provide a list of containers when creating an automatic storage table space. Instead, the database manager automatically creates containers on all of the storage paths associated with the database. Data is striped evenly across all containers so that the storage paths are used equally. | Requires that containers be defined as a directory name. | • Requires that containers be defined as files or devices.<br>• Must specify the initial size for each container. |
| Initial allocation of space | • For nontemporary automatic storage table spaces:<br>– Space is allocated when the table space is created<br>– You can specify the initial size for table space<br>• For temporary automatic storage table spaces, space is allocated as needed. | Done as needed. Because the file system controls the allocation of storage, there is less likelihood that pages will be contiguous, which could have an impact on the performance of some types of queries. | Done when table space created.<br>• Extents are more likely to be contiguous than they would be with SMS table spaces.<br>• Pages within extents are always contiguous for device containers. |
| Changes to table space containers | • Containers can dropped or reduced if the table space size is reduced.<br>• Table space can be rebalanced to distribute data evenly across containers when new storage is added to or dropped from the database. | No changes once created, other than to add containers for new data partitions as they are added. | • Containers can be extended or added. A rebalance of the table space data will occur if the new space is added below the high water mark for the table space.<br>• Containers can be reduced or dropped. A rebalance will occur if there is data in the space being dropped |

*Table 8. Comparison of SMS, DMS and automatic storage table spaces (continued)*

| | **Automatic storage table spaces** | **SMS table spaces** | **DMS table spaces** |
|---|---|---|---|
| Handling of demands for increased storage | • Containers are extended automatically up to constraints imposed by file system.<br>• If storage paths are added to the database, containers are extended or created automatically. | Containers will grow until they reach the capacity imposed by the file system. The table space is considered to be full when any one container reaches its maximum capacity. | Containers can be extended beyond the initially-allocated size manually or automatically (if auto-resize is enabled) up to constraints imposed by file system. |
| Ability to place different types of objects in different table spaces | Tables, storage for related large objects (LOBs) and indexes can each reside in separate table spaces. | For partitioned tables only, indexes and index partitions can reside in a table space separate from the one containing table data. | Tables, storage for related large objects (LOBs) and indexes can each reside in separate table spaces. |
| Ongoing maintenance requirements | • Reducing size of table space<br>• Lowering high water mark<br>• Rebalancing | None | • Adding or extending containers<br>• Dropping or reducing containers<br>• Lowering high water mark<br>• Rebalancing |
| Use of restore to redefine containers | You cannot use a redirected restore operation to redefine the containers associated with the table space because the database manager manages space. | You can use a redirected restore operation to redefine the containers associated with the table space | You can use a redirected restore operation to redefine the containers associated with the table space |
| Performance | Similar to DMS | Generally slower than DMS and automatic storage, especially for larger tables. | Generally superior to SMS |

Automatic storage table spaces are the easiest table spaces to set up and maintain, and are recommended for most applications. They are particularly beneficial when:

- You have larger tables or tables that are likely to grow quickly
- You do not want to have to make regular decisions about how to manage container growth.
- You want to be able to store different types of related objects (for example, tables, LOBs, indexes) in different table spaces to enhance performance.

## Defining initial table spaces on database creation

When a database is created, three table spaces are defined by default. The SYSCATSPACE for the system catalog tables. The TEMPSPACE1 for system temporary tables created during database processing. The USERSPACE1 for user-defined tables and indexes. You can also specify additional user table spaces or characteristics for the default table spaces to be created at the database creation.

### About this task

**Note:** When you first create a database no user temporary table space is created.

Unless otherwise specified, the three default table spaces are managed by automatic storage.

Using the **CREATE DATABASE** command, you can specify the page size for the default buffer pool and the initial table spaces. This default also represents the default page size for all future CREATE BUFFERPOOL and CREATE TABLESPACE statements. If you do not specify the page size when creating the database, the default page size is 4 KB.

To define initial table spaces using the command line, enter:

```
CREATE DATABASE name
  PAGESIZE page size
  CATALOG TABLESPACE
    MANAGED BY AUTOMATIC STORAGE
    EXTENTSIZE value PREFETCHSIZE value
  USER TABLESPACE
    MANAGED BY AUTOMATIC STORAGE
    EXTENTSIZE value PREFETCHSIZE value
  TEMPORARY TABLESPACE
    MANAGED BY AUTOMATIC STORAGE
  WITH "comment"
```

If you do not want to use the default definition for these table spaces, you might specify their characteristics on the **CREATE DATABASE** command. For example, the following command could be used to create your database on Windows:

```
CREATE DATABASE PERSONL
  PAGESIZE 16384
  CATALOG TABLESPACE
    MANAGED BY AUTOMATIC STORAGE
    EXTENTSIZE 16 PREFETCHSIZE 32
  USER TABLESPACE
    MANAGED BY AUTOMATIC STORAGE
    EXTENTSIZE 32 PREFETCHSIZE 64
  TEMPORARY TABLESPACE
    MANAGED BY AUTOMATIC STORAGE
  WITH "Personnel DB for BSchiefer Co"
```

In this example, the default page size is set to 16 384 bytes, and the definition for each of the initial table spaces is explicitly provided. You only need to specify the table space definitions for those table spaces for which you do not want to use the default definition.

**Note:** When working in a partitioned database environment, you cannot create or assign containers to specific database partitions. First, you must create the database with default user and temporary table spaces. Then you should use the CREATE TABLESPACE statement to create the required table spaces. Finally, you can drop the default table spaces.

The coding of the MANAGED BY phrase on the **CREATE DATABASE** command follows the same format as the MANAGED BY phrase on the CREATE TABLESPACE statement.

You can add additional user and temporary table spaces if you want. You cannot drop the catalog table space SYSCATSPACE, or create another one; and there must always be at least one system temporary table space with a page size of 4 KB. You can create other system temporary table spaces. You also cannot change the page size or the extent size of a table space after it has been created.

# Chapter 18. Schemas

A *schema* is a collection of named objects; it provides a way to group those objects logically. A schema is also a name qualifier; it provides a way to use the same natural name for several objects, and to prevent ambiguous references to those objects.

For example, the schema names 'INTERNAL' and 'EXTERNAL' make it easy to distinguish two different SALES tables (INTERNAL.SALES, EXTERNAL.SALES).

Schemas also enable multiple applications to store data in a single database without encountering namespace collisions.

A schema is distinct from, and should not be confused with, an *XML schema*, which is a standard that describes the structure and validates the content of XML documents.

A schema can contain tables, views, nicknames, triggers, functions, packages, and other objects. A schema is itself a database object. It is explicitly created using the CREATE SCHEMA statement, with the current user or a specified authorization ID recorded as the schema owner. It can also be implicitly created when another object is created, if the user has IMPLICIT_SCHEMA authority.

A *schema name* is used as the high order part of a two-part object name. If the object is specifically qualified with a schema name when created, the object is assigned to that schema. If no schema name is specified when the object is created, the default schema name is used (specified in the CURRENT SCHEMA special register).

For example, a user with DBADM authority creates a schema called C for user A:

```
CREATE SCHEMA C AUTHORIZATION A
```

User A can then issue the following statement to create a table called X in schema C (provided that user A has the CREATETAB database authority):

```
CREATE TABLE C.X (COL1 INT)
```

Some schema names are reserved. For example, built-in functions belong to the SYSIBM schema, and the pre-installed user-defined functions belong to the SYSFUN schema.

When a database is created, if it is not created with the `RESTRICTIVE` option, all users have IMPLICIT_SCHEMA authority. With this authority, users implicitly create a schema whenever they create an object with a schema name that does not already exist. When schemas are implicitly created, CREATEIN privileges are granted which allows any user to create other objects in this schema. The ability to create objects such as aliases, distinct types, functions, and triggers is extended to implicitly created schemas. The default privileges on an implicitly created schema provide backward compatibility with previous versions.

The owner of an implicitly created schema is SYSIBM. When the database is restrictive, PUBLIC does not have the CREATEIN privilege on the schema. The

user who implicitly creates the schema has CREATEIN privilege on the schema. When the database is not restrictive, PUBLIC has the CREATEIN privilege on the schema.

If IMPLICIT_SCHEMA authority is revoked from PUBLIC, schemas can be explicitly created using the CREATE SCHEMA statement, or implicitly created by users (such as those with DBADM authority) who have been granted IMPLICIT_SCHEMA authority. Although revoking IMPLICIT_SCHEMA authority from PUBLIC increases control over the use of schema names, it can result in authorization errors when existing applications attempt to create objects.

Schemas also have privileges, allowing the schema owner to control which users have the privilege to create, alter, and drop objects in the schema. This ability provides a way to control the manipulation of a subset of objects in the database. A schema owner is initially given all of these privileges on the schema, with the ability to grant the privileges to others. An implicitly created schema is owned by the system, and all users are initially given the privilege to create objects in such a schema, except in a restrictive database environment. A user with ACCESSCTRL or SECADM authority can change the privileges that are held by users on any schema. Therefore, access to create, alter, and drop objects in any schema (even one that was implicitly created) can be controlled.

## Schema name restrictions and recommendations

There are some restrictions and recommendations that you must be aware of when naming schemas.

- User-defined types (UDTs) cannot have schema names longer than the schema length listed in "SQL and XML limits" in the *SQL Reference*.
- The following schema names are reserved words and must not be used: SYSCAT, SYSFUN, SYSIBM, SYSSTAT, SYSPROC.
- To avoid potential problems upgrading databases in the future, do not use schema names that begin with SYS. The database manager will not allow you to create modules, procedures, triggers, user-defined types or user-defined functions using a schema name beginning with SYS.
- It is recommended that you not use SESSION as a schema name. Declared temporary tables must be qualified by SESSION. It is therefore possible to have an application declare a temporary table with a name identical to that of a persistent table, in which case the application logic can become overly complicated. Avoid the use of the schema SESSION, except when dealing with declared temporary tables.

## Creating schemas

You can use schemas to group objects as you create those objects. An object can belong to only one schema. Use the CREATE SCHEMA statement to create schemas.

Information about the schemas is kept in the system catalog tables of the database to which you are connected.

### Before you begin

To create a schema and optionally make another user the owner of the schema, you need DBADM authority. If you do not hold DBADM authority, you can still create a schema using your own authorization ID. The definer of any objects

created as part of the CREATE SCHEMA statement is the schema owner. This owner can GRANT and REVOKE schema privileges to other users.

### Procedure

To create a schema from the command line, enter the following statement:

```
CREATE SCHEMA schema-name [ AUTHORIZATION schema-owner-name ]
```

Where *schema-name* is the name of the schema. This name must be unique within the schemas already recorded in the catalog, and the name cannot begin with SYS. If the optional AUTHORIZATION clause is specified, the *schema-owner-name* becomes the owner of the schema. If this clause is not specified, the authorization ID that issued this statement becomes the owner of the schema.
For more information, see the CREATE SCHEMA statement. See also "Schema name restrictions and recommendations" on page 106.

## Dropping schemas

To delete a schema, use the DROP statement.

### Before you begin

Before dropping a schema, all objects that were in that schema must be dropped or moved to another schema.

The schema name must be in the catalog when attempting the DROP statement; otherwise an error is returned.

### Procedure

To drop a schema by using the command line, enter:

```
DROP SCHEMA name RESTRICT
```

The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database. The RESTRICT keyword is not optional.

### Example

In the following example, the schema "joeschma" is dropped:

```
DROP SCHEMA joeschma RESTRICT
```

# Chapter 19. Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows.

At the intersection of every column and row is a specific data item called a *value*. A *column* is a set of values of the same type or one of its subtypes. A *row* is a sequence of values arranged so that the *n*th value is a value of the *n*th column of the table.

An application program can determine the order in which the rows are populated into the table, but the actual order of rows is determined by the database manager, and typically cannot be controlled. Multidimensional clustering (MDC) provides some sense of clustering, but not actual ordering between the rows.

## Types of tables

DB2 databases store data in tables. In addition to tables used to store persistent data, there are also tables that are used for presenting results, summary tables and temporary tables; multidimensional clustering tables offer specific advantages in a warehouse environment.

**Base tables**

These types of tables hold persistent data. There are different kinds of base tables, including

**Regular tables**

Regular tables with indexes are the "general purpose" table choice.

**Multidimensional clustering (MDC) tables**

These types of tables are implemented as tables that are physically clustered on more than one key, or dimension, at the same time. MDC tables are used in data warehousing and large database environments. Clustering indexes on regular tables support single-dimensional clustering of data. MDC tables provide the benefits of data clustering across more than one dimension. MDC tables provide *guaranteed clustering* within the composite dimensions. By contrast, although you can have a clustered index with regular tables, clustering in this case is attempted by the database manager, but not guaranteed and it typically degrades over time. MDC tables can coexist with partitioned tables and can themselves be partitioned tables.

Multidimensional clustering tables are not supported in a DB2 pureScale environment.

**Insert time clustering (ITC) tables**

These types of tables are conceptually, and physically similar to MDC tables, but rather than being clustered by one or more user specified dimensions, rows are clustered by the time they are inserted into the table. ITC tables can be partitioned tables.

ITC tables are not supported in a DB2 pureScale environment.

**Range-clustered tables (RCT)**

These types of tables are implemented as sequential clusters of data that provide fast, direct access. Each record in the table has a

predetermined record ID (RID) which is an internal identifier used to locate a record in a table. RCT tables are used where the data is tightly clustered across one or more columns in the table. The largest and smallest values in the columns define the range of possible values. You use these columns to access records in the table; this is the most optimal method of using the predetermined record identifier (RID) aspect of RCT tables.

Range-clustered tables are not supported in a DB2 pureScale environment.

**Partitioned tables**

These types of tables use a data organization scheme in which table data is divided across multiple storage objects, called data partitions or ranges, according to values in one or more table partitioning key columns of the table. Data partitions can be added to, attached to, and detached from a partitioned table, and you can store multiple data partition ranges from a table in one table space. Partitioned tables can contain large amounts of data and simplify the rolling in and rolling out of table data.

**Temporal tables**

These types of tables are used to associate time-based state information to your data. Data in tables that do not use temporal support represents the present, while data in temporal tables is valid for a period defined by the database system, customer applications, or both. For example, a database can store the history of a table (deleted rows or the original values of rows that have been updated) so you can query the past state of your data. You can also assign a date range to a row of data to indicate when it is deemed to be valid by your application or business rules.

**Temporary tables**

These types of tables are used as temporary work tables for various database operations. *Declared temporary tables* (DGTTs) do not appear in the system catalog, which makes them not persistent for use by, and not able to be shared with other applications. When the application using this table terminates or disconnects from the database, any data in the table is deleted and the table is dropped. By contrast, *created temporary tables* (CGTTs) do appear in the system catalog and are not required to be defined in every session where they are used. As a result, they are persistent and able to be shared with other applications across different connections.

Neither type of temporary table supports

- User-defined reference or user-defined structured type columns
- LONG VARCHAR columns

In addition XML columns cannot be used in created temporary tables.

**Materialized query tables**

These types of tables are defined by a query that is also used to determine the data in the table. Materialized query tables can be used to improve the performance of queries. If the database manager determines that a portion of a query can be resolved using a summary table, the database manager can rewrite the query to use the summary table. This decision is based on database configuration settings, such as the CURRENT REFRESH AGE and the CURRENT QUERY OPTIMIZATION special registers. A summary table is a specialized type of materialized query table.

You can create all of the preceding types of tables using the CREATE TABLE statement.

Depending on what your data is going to look like, you might find one table type offers specific capabilities that can optimize storage and query performance. For example, if you have data records that are loosely clustered (not monotonically increasing), consider using a regular table and indexes. If you have data records that have duplicate (but not unique) values in the key, do not use a range-clustered table. Also, if you cannot afford to preallocate a fixed amount of storage on disk for the range-clustered tables you might want, do not use this type of table. If you have data that has the potential for being clustered along multiple dimensions, such as a table tracking retail sales by geographic region, division and supplier, a multidimensional clustering table might suit your purposes.

In addition to the various table types described previously, you also have options for such characteristics as *partitioning*, which can improve performance for tasks such as rolling in table data. Partitioned tables can also hold much more information than a regular, nonpartitioned table. You can also use capabilities such as *compression*, which can help you significantly reduce your data storage costs.

## Data organization schemes

With the introduction of table partitioning, a DB2 database offers a three-level data organization scheme. There are three clauses of the CREATE TABLE statement that include an algorithm to indicate how the data is to be organized.

The following three clauses demonstrate the levels of data organization that can be used together in any combination:

- DISTRIBUTE BY to spread data evenly across database partitions (to enable intraquery parallelism and to balance the load across each database partition) (database partitioning)
- PARTITION BY to group rows with similar values of a single dimension in the same data partition (table partitioning)
- ORGANIZE BY to group rows with similar values on multiple dimensions in the same table extent (multidimensional clustering) or to group rows according to the time of the insert operation (insert time clustering table).

This syntax allows consistency between the clauses and allows for future algorithms of data organization. Each of these clauses can be used in isolation or in combination with one another. By combining the DISTRIBUTE BY and PARTITION BY clauses of the CREATE TABLE statement data can be spread across database partitions spanning multiple table spaces. This approach allows for similar behavior to the Informix Dynamic Server and Informix Extended Parallel Server hybrid.

In a single table, you can combined the clauses used in each data organization scheme to create more sophisticated partitioning schemes. For example, partitioned database environments are not only compatible, but also complementary to table partitioning.

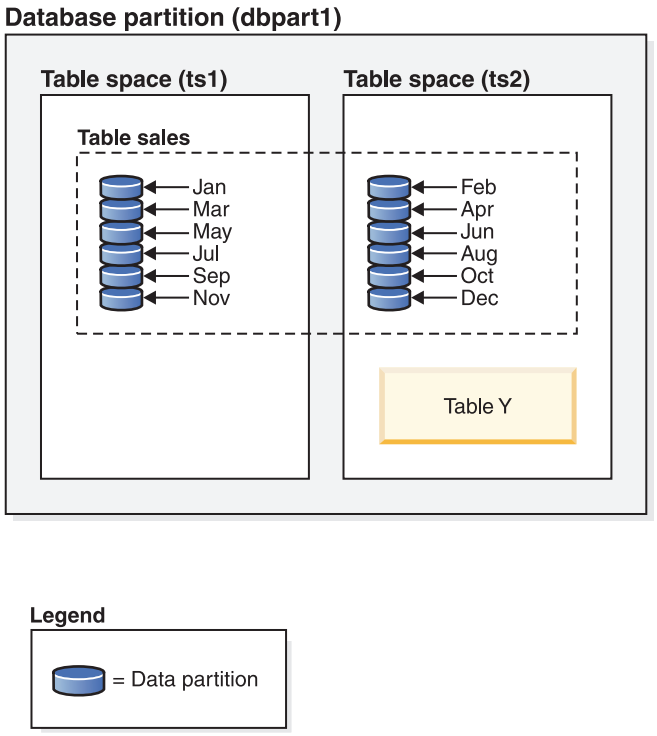



*Figure 7. Demonstrating the table partitioning organization scheme where a table representing monthly sales data is partitioned into multiple data partitions. The table also spans two table spaces (ts1 and ts2).*

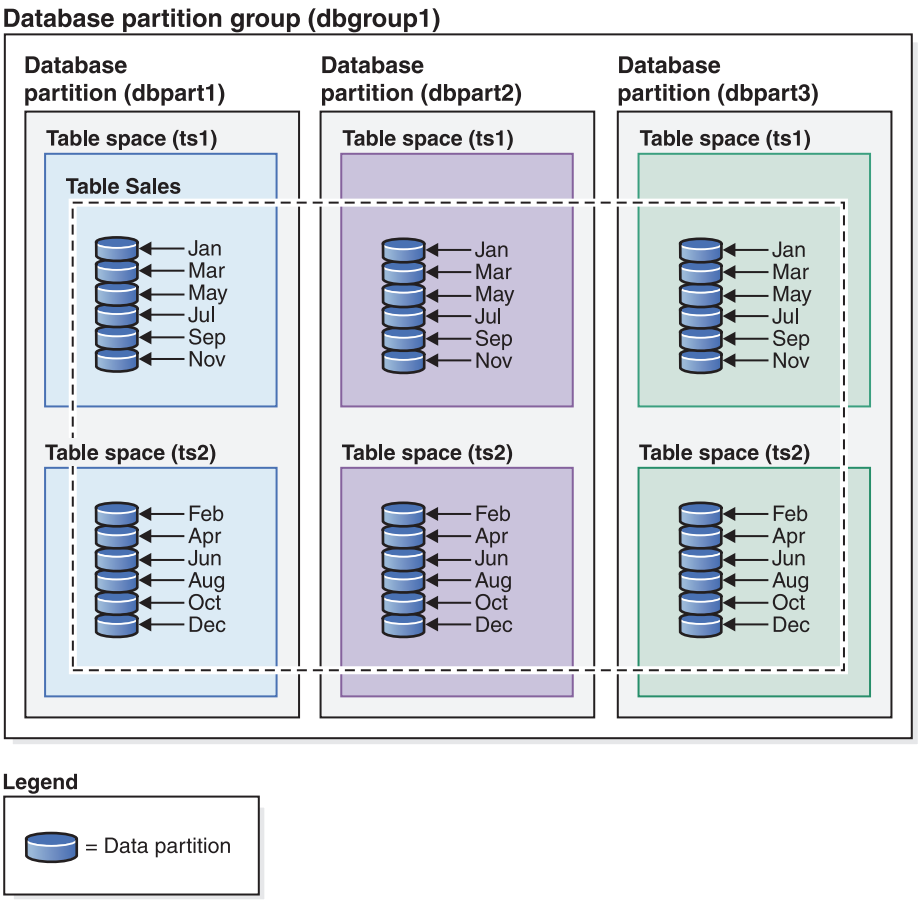



*Figure 8. Demonstrating the complementary organization schemes of database partitioning and table partitioning. A table representing monthly sales data is partitioned into multiple data partitions, spanning two table spaces (ts1 and ts2) that are distributed across multiple database partitions (dbpart1, dbpart2, dbpart3) of a database partition group (dbgroup1).*

The salient distinction between multidimensional clustering (MDC) and table partitioning is multi-dimension versus single dimension. MDC is suitable to cubes (that is, tables with multiple dimensions), and table partitioning works well if there is a single dimension which is central to the database design, such as a DATE column. MDC and table partitioning are complementary when both of these conditions are met. This is demonstrated in Figure 9 on page 114.

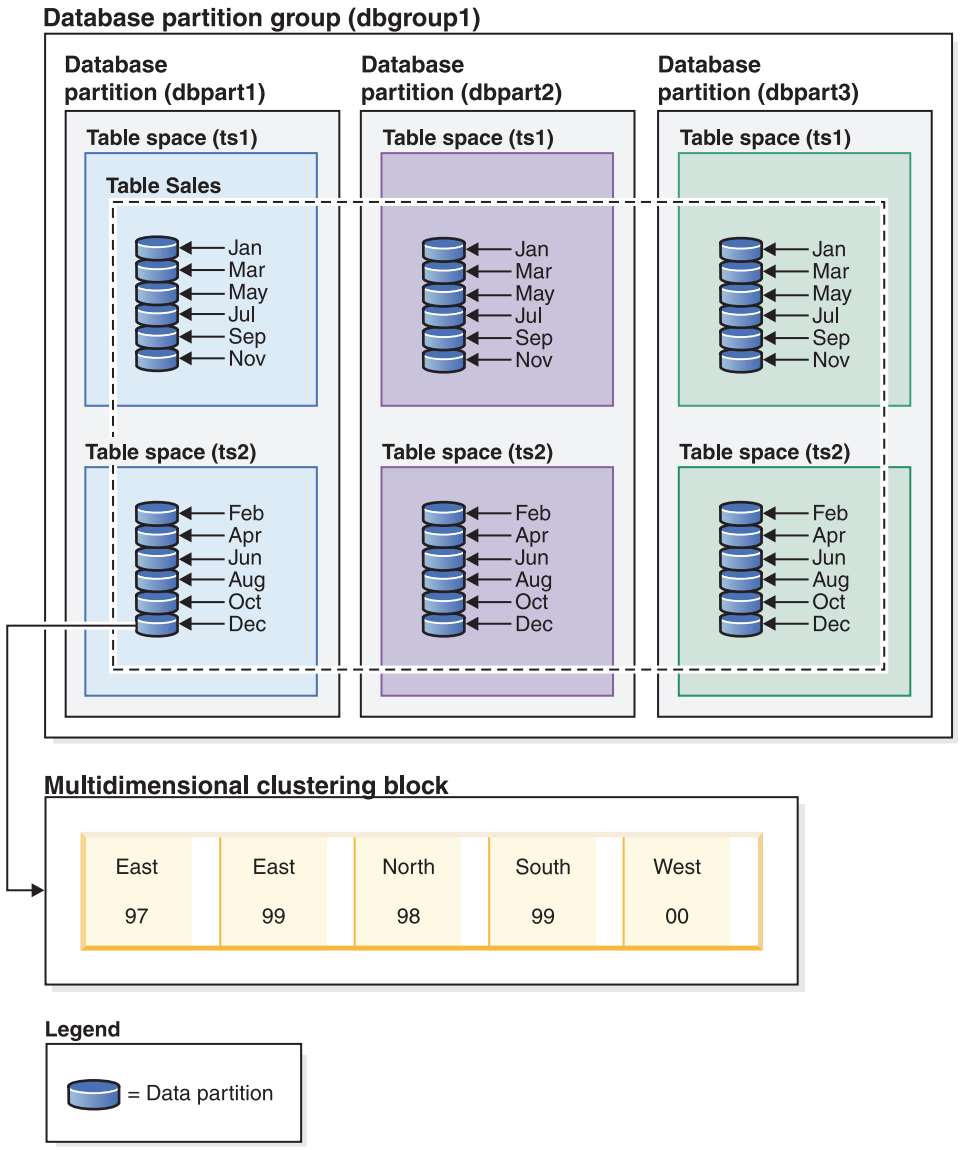



*Figure 9. A representation of the database partitioning, table partitioning and multidimensional organization schemes where data from table SALES is not only distributed across multiple database partitions, partitioned across table spaces ts1 and ts2, but also groups rows with similar values on both the date and region dimensions.*

There is another data organization scheme which cannot be used with any of the schemes that were listed previously. This scheme is ORGANIZE BY KEY SEQUENCE. It is used to insert each record into a row that was reserved for that record at the time of table creation (Range-clustered table).

## Data organization terminology

**Database partitioning**

A data organization scheme in which table data is divided across multiple database partitions based on the hash values in one or more distribution key columns of the table, and based on the use of a distribution map of the database partitions. Data from a given table is distributed based on the specifications provided in the DISTRIBUTE BY HASH clause of the CREATE TABLE statement.

**Database partition**
A portion of a database on a database partition server consisting of its own user data, indexes, configuration file, and transaction logs. Database partitions can be logical or physical.

**Table partitioning**
A data organization scheme in which table data is divided across multiple data partitions according to values in one or more partitioning columns of the table. Data from a given table is partitioned into multiple storage objects based on the specifications provided in the PARTITION BY clause of the CREATE TABLE statement. These storage objects can be in different table spaces.

**Data partition**
A set of table rows, stored separately from other sets of rows, grouped by the specifications provided in the PARTITION BY RANGE clause of the CREATE TABLE statement.

**Multidimensional clustering (MDC)**
A table whose data is physically organized into blocks along one or more dimensions, or clustering keys, specified in the ORGANIZE BY DIMENSIONS clause.

**Insert time clustering (ITC)**
A table whose data is physically clustered based on row insert time, specified by the ORGANIZE BY INSERT TIME clause.

## Benefits of each data organization scheme

Understanding the benefits of each data organization scheme can help you to determine the best approach when planning, designing, or reassessing your database system requirements. Table 9 provides a high-level view of common customer requirements and shows how the various data organization schemes can help you to meet those requirements.

*Table 9. Using table partitioning with the Database Partitioning Feature*

| Issue | Recommended scheme | Explanation |
|---|---|---|
| Data roll-out | Table partitioning | Uses detach to roll out large amounts of data with minimal disruption |
| Parallel query execution (query performance) | Database Partitioning Feature | Provides query parallelism for improved query performance |
| Data partition elimination (query performance) | Table partitioning | Provides data partition elimination for improved query performance |
| Maximization of query performance | Both | Maximum query performance when used together: query parallelism and data partition elimination are complementary |
| Heavy administrator workload | Database Partitioning Feature | Execute many tasks for each database partition |

*Table 10. Using table partitioning with MDC tables*

| Issue | Recommended scheme | Explanation |
|---|---|---|
| Data availability during roll-out | Table partitioning | Use the DETACH PARTITION clause to roll out large amounts of data with minimal disruption. |
| Query performance | Both | MDC is best for querying multiple dimensions. Table partitioning helps through data partition elimination. |
| Minimal reorganization | MDC | MDC maintains clustering, which reduces the need to reorganize. |

**Note:** Table partitioning is now recommended over UNION ALL views.

## Data types and table columns

When you create your table, you must indicate what type of data each column will store. By thinking carefully about the nature of the data you are going to be managing, you can set your tables up in a way that will give you optimal query performance, minimize physical storage requirements, and provide you with specialized capabilities for manipulating different kinds of data, such as arithmetic operations for numeric data, or comparing date or time values to one another.

Figure 10 on page 117 shows the data types that are supported by DB2 databases.

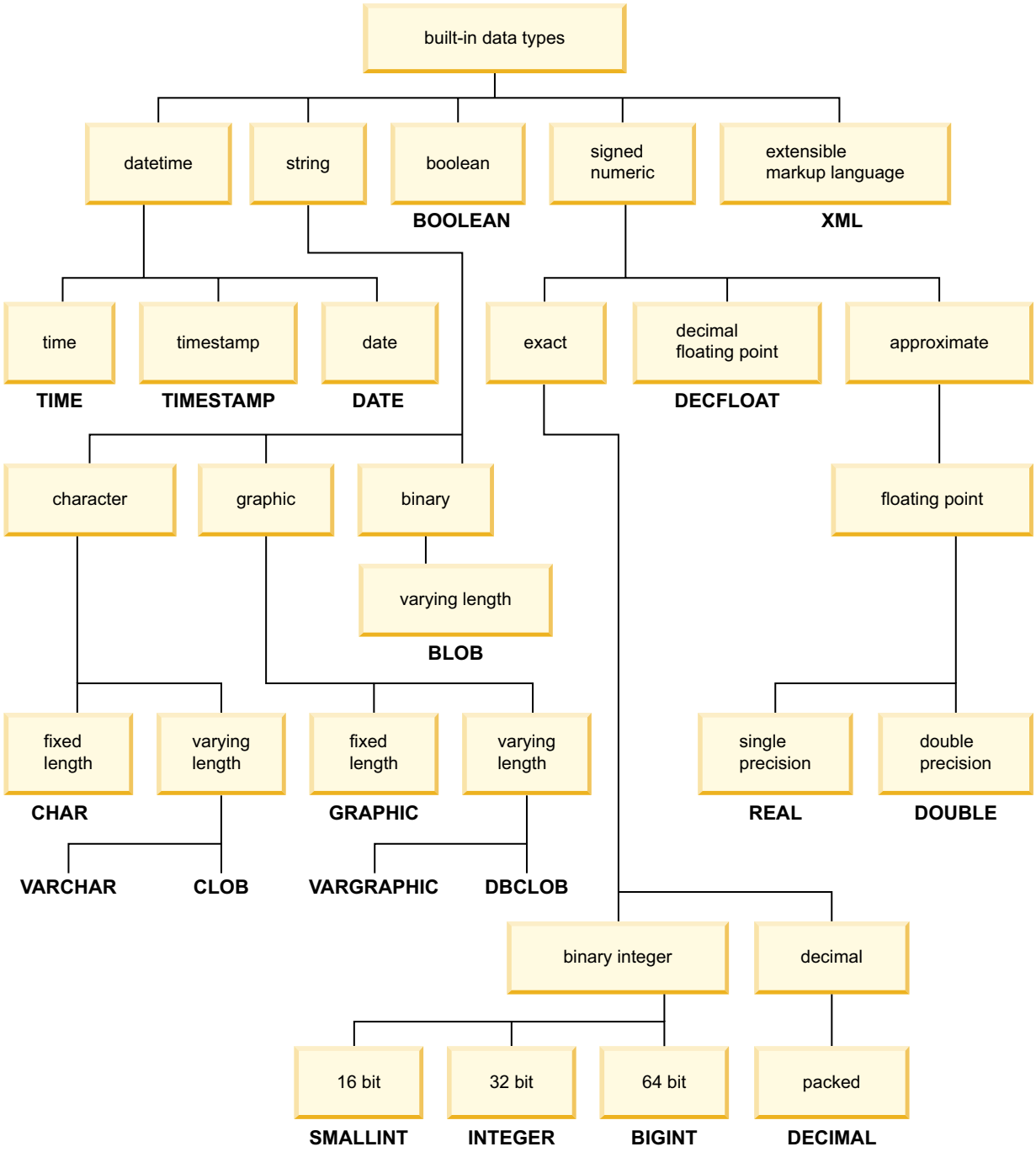



*Figure 10. Built-in data types*

When you declare your database columns all of these data tyoes are available for you to choose from. In addition to the built-in types, you can also create your own *user-defined* data types that are based on the built-in types. For example, if you might choose to represent an employee with name, job title, job level, hire date and salary attributes with a user-defined *structured* type that incorporates VARCHAR (name, job title), SMALLINT (job level), DATE (hire date) and DECIMAL (salary) data.

## Numbers

The numeric data types are integer, decimal, floating-point, and decimal floating-point.

The numeric data types are categorized as follows:

- Exact numerics: integer and decimal
- Decimal floating-point
- Approximate numerics: floating-point

Integer includes small integer, large integer, and big integer. Integer numbers are exact representations of integers. Decimal numbers are exact representations of numbers with a fixed precision and scale. Integer and decimal numbers are considered exact numeric types.

Decimal floating-point numbers can have a precision of 16 or 34. Decimal floating-point supports both exact representations of real numbers and approximation of real numbers and so is not considered either an exact numeric type or an approximate numeric type.

Floating-point includes single precision and double precision. Floating-point numbers are approximations of real numbers and are considered approximate numeric types.

All numbers have a *sign*, a *precision*, and a *scale*. For all numbers except decimal floating-point, if a column value is zero, the sign is positive. Decimal floating-point numbers include negative and positive zeros. Decimal floating-point has distinct values for a number and the same number with various exponents (for example: 0.0, 0.00, 0.0E5, 1.0, 1.00, 1.0000). The precision is the total number of decimal digits, excluding the sign. The scale is the total number of decimal digits to the right of the decimal point. If there is no decimal point, the scale is zero.

See also the data type section in the description of the CREATE TABLE statement.

### Small integer (SMALLINT)

A *small integer* is a two-byte integer with a precision of 5 digits. The range of small integers is -32 768 to 32 767.

### Large integer (INTEGER)

A *large integer* is a four-byte integer with a precision of 10 digits. The range of large integers is -2 147 483 648 to +2 147 483 647.

### Big integer (BIGINT)

A *big integer* is an eight-byte integer with a precision of 19 digits. The range of big integers is -9 223 372 036 854 775 808 to +9 223 372 036 854 775 807.

### Decimal (DECIMAL or NUMERIC)

A *decimal* value is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits.

All values in a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is *-n* to *+n*, where the absolute value of *n* is the largest number that can be represented with the applicable precision and scale. The maximum range is $-10^{31}+1$ to $10^{31}-1$.

## Single-precision floating-point (REAL)

A *single-precision floating-point* number is a 32-bit approximation of a real number. The number can be zero or can range from -3.4028234663852886e+38 to -1.1754943508222875e-38, or from 1.1754943508222875e-38 to 3.4028234663852886e+38.

## Double-precision floating-point (DOUBLE or FLOAT)

A *double-precision floating-point* number is a 64-bit approximation of a real number. The number can be zero or can range from -1.7976931348623158e+308 to -2.2250738585072014e-308, or from 2.2250738585072014e-308 to 1.7976931348623158e+308.

## Decimal floating-point (DECFLOAT)

A *decimal floating-point* value is an IEEE 754r number with a decimal point. The position of the decimal point is stored in each decimal floating-point value. The maximum precision is 34 digits. The range of a decimal floating-point number is either 16 or 34 digits of precision, and an exponent range of $10^{-383}$ to $10^{+384}$ or $10^{-6143}$ to $10^{+6144}$, respectively. The minimum exponent, $E_{min}$, for DECFLOAT values is -383 for DECFLOAT(16) and -6143 for DECFLOAT(34). The maximum exponent, $E_{max}$, for DECFLOAT values is 384 for DECFLOAT(16) and 6144 for DECFLOAT(34).

In addition to finite numbers, decimal floating-point numbers are able to represent one of the following named decimal floating-point special values:

- Infinity - a value that represents a number whose magnitude is infinitely large
- Quiet NaN - a value that represents undefined results and that does not cause an invalid number warning
- Signalling NaN - a value that represents undefined results and that causes an invalid number warning if used in any numeric operation

When a number has one of these special values, its coefficient and exponent are undefined. The sign of an infinity value is significant, because it is possible to have positive or negative infinity. The sign of a NaN value has no meaning for arithmetic operations.

## Subnormal numbers and underflow

Nonzero numbers whose adjusted exponents are less than $E_{min}$ are called subnormal numbers. These subnormal numbers are accepted as operands for all operations and can result from any operation.

For a subnormal result, the minimum values of the exponent become $E_{min}$ - (precision-1), called $E_{tiny}$, where precision is the working precision. If necessary, the result is rounded to ensure that the exponent is no smaller than $E_{tiny}$. If the result becomes inexact during rounding, an underflow warning is returned. A subnormal result does not always return the underflow warning.

When a number underflows to zero during a calculation, its exponent will be $E_{tiny}$. The maximum value of the exponent is unaffected.

The maximum value of the exponent for subnormal numbers is the same as the minimum value of the exponent that can arise during operations that do not result in subnormal numbers. This occurs when the length of the coefficient in decimal digits is equal to the precision.

## Character strings

A *character string* is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the *empty string*. This value should not be confused with the null value.

### Fixed-length character string (CHAR)

All values in a fixed-length string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 254, inclusive.

### Varying-length character strings

There are two types of varying-length character strings:

- A VARCHAR value can be up to 32 672 bytes long.
- A CLOB (character large object) value can be up to 2 gigabytes minus 1 byte (2 147 483 647 bytes) long. A CLOB is used to store large SBCS or mixed (SBCS and MBCS) character-based data (such as documents written with a single character set) and, therefore, has an SBCS or mixed code page associated with it.

Special restrictions apply to expressions resulting in a CLOB data type, and to structured type columns; such expressions and columns are not permitted in:

- A SELECT list preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, or IN predicate
- An aggregate function
- VARGRAPHIC, TRANSLATE, and datetime scalar functions
- The pattern operand in a LIKE predicate, or the search string operand in a POSSTR function
- The string representation of a datetime value.

The functions in the SYSFUN schema taking a VARCHAR as an argument will not accept VARCHARs greater than 4 000 bytes long as an argument. However, many of these functions also have an alternative signature accepting a CLOB(1M). For these functions, the user can explicitly cast the greater than 4 000 VARCHAR strings into CLOBs and then recast the result back into VARCHARs of the required length.

NUL-terminated character strings found in C are handled differently, depending on the standards level of the precompile option.

Each character string is further defined as one of:

**Bit data**
Data that is not associated with a code page.

**Single-byte character set (SBCS) data**
Data in which every character is represented by a single byte.

**Mixed data**
Data that may contain a mixture of characters from a single-byte character set and a multi-byte character set (MBCS).

**Note:** The LONG VARCHAR data type continues to be supported but is deprecated, not recommended, and might be removed in a future release.

## String units in built-in functions

The ability to specify string units for certain built-in functions allows you to process string data in a more "character-based manner" than a "byte-based manner". The *string unit* determines the length in which an operation is to occur. You can specify CODEUNITS16, CODEUNITS32, or OCTETS as the string unit for an operation.

**CODEUNITS16**
Specifies that Unicode UTF-16 is the unit for the operation. CODEUNITS16 is useful when an application is processing data in code units that are two bytes in width. Note that some characters, known as *supplementary characters*, require two UTF-16 code units to be encoded. For example, the musical symbol G clef requires two UTF-16 code units (X'D834' and X'DD1E' in UTF-16BE).

**CODEUNITS32**
Specifies that Unicode UTF-32 is the unit for the operation. CODEUNITS32 is useful for applications that process data in a simple, fixed-length format, and that must return the same answer regardless of the storage format of the data (ASCII, UTF-8, or UTF-16).

**OCTETS**
Specifies that bytes are the units for the operation. OCTETS is often used when an application is interested in allocating buffer space or when operations need to use simple byte processing.

The calculated length of a string computed using OCTETS (bytes) might differ from that computed using CODEUNITS16 or CODEUNITS32. When using OCTETS, the length of the string is determined by simply counting the number of bytes in the string. When using CODEUNITS16 or CODEUNITS32, the length of the string is determined by counting the number of 16-bit or 32-bit code units necessary to represent the string in UTF-16 or UTF-32, respectively. The length determined using CODEUNITS16 and CODEUNITS32 will be identical unless the data contains supplementary characters.

For example, assume that NAME, a VARCHAR(128) column encoded in Unicode UTF-8, contains the value 'Jürgen'. The following two queries, which count the length of the string in CODEUNITS16 and CODEUNITS32, respectively, return the same value (6).

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS16) FROM T1
  WHERE NAME = 'Jürgen'

SELECT CHARACTER_LENGTH(NAME,CODEUNITS32) FROM T1
  WHERE NAME = 'Jürgen'
```

The next query, which counts the length of the string in OCTETS, returns the value 7.

```
SELECT CHARACTER_LENGTH(NAME,OCTETS) FROM T1
  WHERE NAME = 'Jürgen'
```

These values represent the length of the string expressed in the specified string unit.

The following table shows the UTF-8, UTF-16BE (big-endian), and UTF-32BE (big-endian) representations of the name 'Jürgen':

```
Format    Representation of the name 'Jürgen'
--------  ------------------------------------
UTF-8   X'4AC3BC7267656E'
UTF-16BE  X'004A00FC007200670065006E'
UTF-32BE  X'0000004A000000FC00000072000000670000006500000006E'
```

The representation of the character 'ü' differs among the three string units:

- The UTF-8 representation of the character 'ü' is X'C3BC'.
- The UTF-16BE representation of the character 'ü' is X'00FC'.
- The UTF-32BE representation of the character 'ü' is X'000000FC'.

Specifying string units for a built-in function does not affect the data type or the code page of the result of the function. If necessary, DB2 converts the data to Unicode for evaluation when CODEUNITS16 or CODEUNITS32 is specified.

When OCTETS is specified for the LOCATE or POSITION function, and the code pages of the string arguments differ, DB2 converts the data to the code page of the *source-string* argument. In this case, the result of the function is in the code page of the *source-string* argument. When OCTETS is specified for functions that take a single string argument, the data is evaluated in the code page of the string argument, and the result of the function is in the code page of the string argument.

## Difference between CODEUNITS16 and CODEUNITS32

When CODEUNITS16 or CODEUNITS32 is specified, the result is the same except when the data contains Unicode supplementary characters. This is because a supplementary character is represented by two UTF-16 code units or one UTF-32 code unit. In UTF-8, a non-supplementary character is represented by 1 to 3 bytes, and a supplementary character is represented by 4 bytes. In UTF-16, a non-supplementary character is represented by one CODEUNITS16 code unit or 2 bytes, and a supplementary character is represented by two CODEUNITS16 code units or 4 bytes. In UTF-32, a character is represented by one CODEUNITS32 code unit or 4 bytes.

For example, the following table shows the hexadecimal values for the mathematical bold capital A and the Latin capital letter A. The mathematical bold capital A is a supplementary character that is represented by 4 bytes in UTF-8, UTF-16, and UTF-32.

| Character | UTF-8 representation | UTF-16BE representation | UTF-32BE representation |
|---|---|---|---|
| Unicode value X'1D400' - 'A'; mathematical bold capital A | X'F09D9080' | X'D835DC00' | X'0001D400' |
| Unicode value X'0041' - 'A'; latin capital letter A | X'41' | X'0041' | X'00000041' |

Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following queries return different results:

```
Query                                                 Returns
-----                                                 -------
SELECT CHARACTER_LENGTH(C1,CODEUNITS16) FROM T1       2

SELECT CHARACTER_LENGTH(C1,CODEUNITS32) FROM T1       1

SELECT CHARACTER_LENGTH(C1,OCTETS) FROM T1            4
```

## Datetime values

The datetime data types include DATE, TIME, and TIMESTAMP. Although datetime values can be used in certain arithmetic and string operations, and are compatible with certain strings, they are neither strings nor numbers.

### Date

A *date* is a three-part value (year, month, and day). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to *x*, where *x* depends on the month.

The internal representation of a date is a string of 4 bytes. Each byte consists of 2 packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column, as described in the SQLDA, is 10 bytes, which is the appropriate length for a character string representation of the value.

### Time

A *time* is a three-part value (hour, minute, and second) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24. The range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications are zero.

The internal representation of a time is a string of 3 bytes. Each byte consists of 2 packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column, as described in the SQLDA, is 8 bytes, which is the appropriate length for a character string representation of the value.

### Timestamp

A *timestamp* is a six or seven-part value (year, month, day, hour, minute, second, and optional fractional seconds) designating a date and time as defined in the previous sections, except that the time could also include an additional part designating a fraction of a second. The number of digits in the fractional seconds is specified using an attribute in the range from 0 to 12 with a default of 6.

The internal representation of a timestamp is a string of between 7 and 13 bytes. Each byte consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 0 to 6 bytes the fractional seconds.

The length of a TIMESTAMP column, as described in the SQLDA, is between 19 and 32 bytes, which is the appropriate length for the character string representation of the value.

## String representations of datetime values

Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the user. Date, time, and timestamp values can, however, also be represented by strings. This is useful because there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Before it can be retrieved, a datetime value must be assigned to a string variable. The CHAR function or the GRAPHIC function (for Unicode databases only) can be used to change a datetime value to a string representation. The string representation is normally the default format of datetime values associated with the territory code of the application, unless overridden by specification of the DATETIME option when the program is precompiled or bound to the database.

No matter what its length, a large object string cannot be used as a string representation of a datetime value (SQLSTATE 42884).

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp value before the operation is performed.

Date, time and timestamp strings must contain only characters and digits.

## Date strings

A string representation of a date is a string that starts with a digit and has a length of at least 8 characters. Trailing blanks may be included; leading zeros may be omitted from the month and day portions.

Valid string formats for dates are listed in the following table. Each format is identified by name and associated abbreviation.

*Table 11. Formats for String Representations of Dates*

| Format Name | Abbreviation | Date Format | Example |
|---|---|---|---|
| International Standards Organization | ISO | yyyy-mm-dd | 1991-10-27 |
| IBM USA standard | USA | mm/dd/yyyy | 10/27/1991 |
| IBM European standard | EUR | dd.mm.yyyy | 27.10.1991 |
| Japanese Industrial Standard Christian Era | JIS | yyyy-mm-dd | 1991-10-27 |
| Site-defined | LOC | Depends on the territory code of the application | - |

## Time strings

A string representation of a time is a string that starts with a digit and has a length of at least 4 characters. Trailing blanks can be included; a leading zero can be omitted from the hour part of the time, and seconds can be omitted entirely. If seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13:30 is equivalent to 13:30:00.

Valid string formats for times are listed in the following table. Each format is identified by name and associated abbreviation.

*Table 12. Formats for String Representations of Times*

| Format Name | Abbreviation | Time Format | Example |
|---|---|---|---|
| International Standards Organization | ISO | hh.mm.ss | 13.30.05 |
| IBM USA standard | USA | hh:mm AM or PM | 1:30 PM |
| IBM European standard | EUR | hh.mm.ss | 13.30.05 |
| Japanese Industrial Standard Christian Era | JIS | hh:mm:ss | 13:30:05 |
| Site-defined | LOC | Depends on the territory code of the application | - |

**Note:**

1. In ISO, EUR, or JIS format, .ss (or :ss) is optional.
2. The International Standards Organization changed the time format so that it is identical to the Japanese Industrial Standard Christian Era format. Therefore, use the JIS format if an application requires the current International Standards Organization format.
3. In the USA time string format, the minutes specification can be omitted, indicating an implicit specification of 00 minutes. Thus, 1 PM is equivalent to 1:00 PM.
4. In the USA time string format, the hour must not be greater than 12 and cannot be 0, except in the special case of 00:00 AM. There is a single space before 'AM' or 'PM'. 'AM' and 'PM' can be represented in lowercase or uppercase characters.

   Using the JIS format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

   - 12:01 AM through 12:59 AM corresponds to 00:01:00 through 00:59:00.
   - 01:00 AM through 11:59 AM corresponds to 01:00:00 through 11:59:00.
   - 12:00 PM (noon) through 11:59 PM corresponds to 12:00:00 through 23:59:00.
   - 12:00 AM (midnight) corresponds to 24:00:00 and 00:00 AM (midnight) corresponds to 00:00:00.

## Timestamp strings

A string representation of a timestamp is a string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss* or *yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn*, where the number of digits for fractional seconds can range from 0 to 12. Trailing blanks may be included. Leading zeros may be omitted from the month, day, and hour part of the timestamp. Trailing zeros can be truncated or entirely omitted from the fractional seconds. If a string representation of a timestamp is implicitly cast to a value with a TIMESTAMP data type, the timestamp precision of the result of the cast is determined by the precision of the TIMESTAMP operand in an expression or the precision of the TIMESTAMP target in an assignment. Digits in the string beyond the timestamp precision of the cast are truncated or any missing digits to reach the timestamp precision of the cast are assumed to be zeros. For example, 1991-3-2-8.30.00 is equivalent to 1991-03-02-08.30.00.000000000000.

A string representation of a timestamp can be given a different timestamp precision by explicitly casting the value to a timestamp with a specified precision. If the string is a constant, an alternative is to precede the string constant with the TIMESTAMP keyword. For example, TIMESTAMP '2007-03-28 14:50:35.123' has the TIMESTAMP(3) data type.

SQL statements also support the ODBC string representation of a timestamp, but as an input value only. The ODBC string representation of a timestamp has the form *yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn*, where the number of digits for fractional seconds can range from 0 to 12..

## Large objects (LOBs)

The term *large object* and the generic acronym LOB refer to the BLOB, CLOB, or DBCLOB data type. In a Unicode database, NCLOB can be used as a synonym for DBCLOB.

LOB values are subject to restrictions, as described in "Varying-length character strings" on page 120. These restrictions apply even if the length attribute of the LOB string is 254 bytes or less.

LOB values can be very large, and the transfer of these values from the database server to client application program host variables can be time consuming. Because application programs typically process LOB values one piece at a time, rather than as a whole, applications can reference a LOB value by using a large object locator.

A *large object locator*, or LOB locator, is a host variable whose value represents a single LOB value on the database server.

An application program can select a LOB value into a LOB locator. Then, using the LOB locator, the application program can request database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, or LENGTH; performing an assignment; searching the LOB with LIKE or POSSTR; or applying user-defined functions against the LOB) by supplying the locator value as input. The resulting output (data assigned to a client host variable) would typically be a small subset of the input LOB value.

LOB locators can represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR( <lob 1> CONCAT <lob 2> CONCAT <lob 3>, <start>, <length> )
```

When a null value is selected into a normal host variable, the indicator variable is set to -1, signifying that the value is null. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Because a locator host variable can itself never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value - the server does not track null values with valid locators.

It is important to understand that a LOB locator represents a value, not a row or a location in the database. Once a value is selected into a locator, there is no operation that one can perform on the original row or table that will affect the value which is referenced by the locator. The value associated with a locator is valid until the transaction ends, or until the locator is explicitly freed, whichever comes first. Locators do not force extra copies of the data to provide this function.

Instead, the locator mechanism stores a description of the base LOB value. The materialization of the LOB value (or expression, as shown previously) is deferred until it is actually assigned to some location - either a user buffer in the form of a host variable, or another record in the database.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. It is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, because a LOB locator is a client representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure used by FETCH, OPEN, or EXECUTE statements.

## XML data type

Use the XML data type to define columns of a table and store XML values. All XML values must be well-formed XML documents. You can use this native data type to store well-formed XML documents in their native hierarchical format in the database alongside other relational data.

XML values are processed in an internal representation that is not a string and not directly comparable to string values. An XML value can be transformed into a serialized string value representing the XML document using the XMLSERIALIZE function or by binding the value to an application variable of an XML, string, or binary type. Similarly, a string value that represents an XML document can be transformed to an XML value using the XMLPARSE function or by binding an application string, binary, or XML application type to an XML value. In SQL data change statements (such as INSERT) involving XML columns, a string or binary value that represents an XML document is transformed into an XML value using an injected XMLPARSE function. An XML value can be implicitly parsed or serialized when exchanged with application string and binary data types.

There is no architectural limit on the size of an XML value in a database. However, note that serialized XML data exchanged with DB2 database server is effectively limited to 2 GB.

XML documents can be inserted, updated and deleted using SQL data manipulation statements. Validation of an XML document against an XML schema, typically performed during insert or update, is supported by the XML schema repository (XSR). The DB2 database system also provides mechanisms for constructing and querying XML values, as well as exporting and importing XML data. An index over XML data can be defined on an XML column, providing improved search performance of XML data. The XML data in table or view columns can be retrieved as serialized string data through various application interfaces.

## Generated columns

A generated column is defined in a table where the stored value is computed using an expression, rather than being specified through an insert or update operation.

When creating a table where it is known that certain expressions or predicates will be used all the time, you can add one or more generated columns to that table. By using a generated column there is opportunity for performance improvements when querying the table data.

For example, there are two ways in which the evaluation of expressions can be costly when performance is important:

1. The evaluation of the expression must be done many times during a query.
2. The computation is complex.

To improve the performance of the query, you can define an additional column that would contain the results of the expression. Then, when issuing a query that includes the same expression, the generated column can be used directly; or, the query rewrite component of the optimizer can replace the expression with the generated column.

Where queries involve the joining of data from two or more tables, the addition of a generated column can allow the optimizer a choice of possibly better join strategies.

Generated columns will be used to improve performance of queries. As a result, generated columns will likely be added after the table has been created and populated.

### Examples

The following is an example of defining a generated column on the CREATE TABLE statement:

```
CREATE TABLE t1 (c1 INT,
                 c2 DOUBLE,
                 c3 DOUBLE GENERATED ALWAYS AS (c1 + c2)
                 c4 GENERATED ALWAYS AS
                    (CASE WHEN c1 > c2 THEN 1 ELSE NULL END))
```

After creating this table, indexes can be created using the generated columns. For example,

```
CREATE INDEX i1 ON t1(c4)
```

Queries can take advantage of the generated columns. For example,

```
SELECT COUNT(*) FROM t1 WHERE c1 > c2
```

can be written as:

```
SELECT COUNT(*) FROM t1 WHERE c4 IS NOT NULL
```

Another example:

```
SELECT c1 + c2 FROM t1 WHERE (c1 + c2) * c1 > 100
```

can be written as:

```
SELECT c3 FROM t1 WHERE c3 * c1 > 100
```

## Hidden columns

When a table column is defined with the implicitly hidden attribute, that column is unavailable unless it is explicitly referenced. For example, if a SELECT * query is run against a table, implicitly hidden columns are not returned in the result table. An implicitly hidden column can always be referenced explicitly wherever a column name can be specified.

In cases where columns and their entries are generated by the database manager, defining such columns as IMPLICITLY HIDDEN can minimize any potential negative impact on your applications. For example, a system-period temporal table

has three columns whose values are generated by the database manager. The database manager uses these columns to preserve historical versions of each table row. Most business applications would work with the historical data, but would rarely work with these three generated columns. Hiding these columns from your applications could reduce application processing time.

When inserting data into a table, an INSERT statement without a column list does not expect values for any implicitly hidden columns. In such cases, if the input includes a value for an implicitly hidden column, that value does not have a corresponding target column and an error is returned (SQLSTATE 42802). Because an INSERT statement without a column list does not include values for implicitly hidden columns, any columns that are defined as implicitly hidden and NOT NULL must have a defined default value

When populating a table with data from an input file, utilities like IMPORT, INGEST, and LOAD require that you specify whether data for the hidden columns is included in the operation. If a column list is not specified, data movement utilities must use the *implicitlyhiddeninclude* or *implicitlyhiddenmissing* file type modifiers when working with tables that contain implicitly hidden columns. You can also use the DB2_DMU_DEFAULT registry variable to set the default behavior when data movement utilities encounter tables with implicitly hidden columns. Similarly, EXPORT requires that you specify whether data for the hidden columns is included in the operation.

The implicitly hidden attribute can be defined on a table column using the CREATE TABLE statement for new tables, or the ALTER TABLE statement for existing tables. If a table is created using a CREATE TABLE statement with the LIKE clause, any implicitly hidden columns in the source table are inherited by the new table. The ALTER TABLE statement can be used to change hidden columns to not hidden or to change not hidden columns to hidden. Altering a table to change the hidden attribute of some columns can impact the behavior of data movement utilities that are working with the table. For example, this might mean that a load operation that ran successfully before the table was altered to define some hidden columns, now returns an error (SQLCODE -2437).

The list of names identifying the columns of a result table from a SELECT query run with the *exposed-name.** option does not include any implicitly hidden columns. A SELECT query run with the `order-by-clause` can include implicitly hidden columns in the *simple-column-name*.

If an implicitly hidden column is explicitly referenced in a materialized query table definition, that column will be a part of the materialized query table. However the column in the materialized query table does not inherit the implicitly hidden attribute. This same behaviour applies to views and tables created with the as-result-table clause.

An implicitly hidden column can be explicitly referenced in a CREATE INDEX statement, ALTER TABLE statement, or in a referential constraint.

A transition variable exists for any column defined as implicitly hidden. In the body of a trigger, a transition variable that corresponds to an implicitly hidden column can be referenced.

Implicitly hidden columns are not supported in created temporary tables and declared temporary tables.

Hidden columns for a table can be displayed using the DESCRIBE command.

```
DESCRIBE TABLE tablename SHOW DETAIL
```

### Example

- *Example 1:* In the following statement, a table is created with an implicitly hidden column.

```
CREATE TABLE CUSTOMER
(
 CUSTOMERNO       INTEGER NOT NULL,
 CUSTOMERNAME     VARCHAR(80),
 PHONENO          CHAR(8) IMPLICITLY HIDDEN
);
```

  A `SELECT *` only returns the column entries for `CUSTOMERNO` and `CUSTOMERNAME`. For example:

```
A123, ACME
B567, First Choice
C345, National Chain
```

  Entries for the `PHONENO` column are hidden unless explicitly referenced.

```
SELECT CUSTOMERNO, CUSTOMERNAME, PHONENO
   FROM CUSTOMER
```

- *Example 2:* If the database table contains implicitly hidden columns, you must specify whether data for the hidden columns is included in data movement operations. The following example uses LOAD to show the different methods to indicate if data for hidden columns is included:
  - Use *insert-column* to explicitly specify the columns into which data is to be inserted.

```
db2 load from delfile1 of del
   insert into table1 (c1, c2, c3,...)
```

  - Use one of the hidden column file type modifiers: specify **implicitlyhiddeninclude** when the input file contains data for the hidden columns, or **implicitlyhiddenmissing** when the input file does not.

```
db2 load from delfile1 of del modified by implicitlyhiddeninclude
   insert into table1
```

  - Use the DB2_DMU_DEFAULT registry variable on the server-side to set the behavior when data movement utilities encounter tables with implicitly hidden columns.

```
db2set DB2_DMU_DEFAULT=IMPLICITLYHIDDENINCLUDE
db2 load from delfile1 of del insert into table1
```

## Auto numbering and identifier columns

An identity column provides a way for DB2 to automatically generate a unique numeric value for each row that is added to the table.

When creating a table in which you must uniquely identify each row that will be added to the table, you can add an identity column to the table. To guarantee a unique numeric value for each row that is added to a table, you should define a unique index on the identity column or declare it a primary key.

Other uses of an identity column are an order number, an employee number, a stock number, or an incident number. The values for an identity column can be generated by the DB2 database manager: ALWAYS or BY DEFAULT.

An identity column defined as GENERATED ALWAYS is given values that are always generated by the DB2 database manager. Applications are not allowed to provide an explicit value. An identity column defined as GENERATED BY DEFAULT gives applications a way to explicitly provide a value for the identity column. If the application does not provide a value, then DB2 will generate one. Since the application controls the value, DB2 cannot guarantee the uniqueness of the value. The GENERATED BY DEFAULT clause is meant for use for data propagation where the intent is to copy the contents of an existing table; or, for the unload and reloading of a table.

Once created, you first have to add the column with the DEFAULT option to get the existing default value. Then you can ALTER the default to become an identity column.

If rows are inserted into a table with explicit identity column values specified, the next internally generated value is not updated, and might conflict with existing values in the table. Duplicate values will generate an error message if the uniqueness of the values in the identity column is being enforced by a primary-key or a unique index that has been defined on the identity column.

To define an identity column on a new table, use the AS IDENTITY clause on the CREATE TABLE statement.

### Example

The following is an example of defining an identity column on the CREATE TABLE statement:

```
CREATE TABLE table (col1 INT,
                    col2 DOUBLE,
                    col3 INT NOT NULL GENERATED ALWAYS AS IDENTITY
                                      (START WITH 100, INCREMENT BY 5))
```

In this example the third column is the identity column. You can also specify the value used in the column to uniquely identify each row when added. Here the first row entered has the value of “100” placed in the column; every subsequent row added to the table has the associated value increased by five.

## Default column and data type definitions

Certain columns and data types have predefined or assigned default values.

For example, default column values for the various data types are as follows:

- *NULL*
- *0* Used for small integer, integer, decimal, single-precision floating point, double-precision floating point, and decimal floating point data type.
- *Blank:* Used for fixed-length and fixed-length double-byte character strings.
- *Zero-length string:* Used for varying-length character strings, binary large objects, character large objects, and double-byte character large objects.
- *Date:* This the system date at the time the row is inserted (obtained from the CURRENT_DATE special register). When a date column is added to an existing table, existing rows are assigned the date January, 01, 0001.
- *Time or Timestamp:* This is the system time or system date/time of the at the time the statement is inserted (obtained from the CURRENT_TIME special register).

When a time column is added to an existing table, existing rows are assigned the time 00:00:00 or a timestamp that contains the date January, 01, 0001 and the time 00:00:00.

**Note:** All the rows get the same default time/timestamp value for a given statement.

- *Distinct user-defined data type:* This is the built-in default value for the base data type of the distinct user-defined data type (cast to the distinct user-defined data type.

# Creating tables

The database manager controls changes and access to the data stored in the tables. You can create tables by using the CREATE TABLE statement.

Complex statements can be used to define all the attributes and qualities of tables. However, if all the defaults are used, the statement to create a table is simple.

## Creating tables like existing tables

Creating a new source table might be necessary when the characteristics of the target table do not sufficiently match the characteristics of the source when issuing the ALTER TABLE statement with the ATTACH PARTITION clause.

Before creating a new source table, you can attempt to correct the mismatch between the existing source table and the target table.

### Before you begin

To create a table, the privileges held by the authorization ID of the statement must include at least one of the following authorities and privileges:

- CREATETAB authority on the database and USE privilege on the table space, as well as one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the table does not exist
  - CREATEIN privilege on the schema, if the schema name of the table refers to an existing schema
- DBADM authority

### About this task

If attempts to correct the mismatch fail, error SQL20408N or SQL20307N is returned.

### Procedure

To create a new source table:

1. Use the **db2look** command to produce the CREATE TABLE statement to create a table identical to the target table:

   ```
   db2look -d source_database_name -t source_table_name -e
   ```

2. Remove the partitioning clause from the **db2look** output and change the name of the table created to a new name (for example, “sourceC”).
3. Next, load all of the data from the original source table to the newly created source table, sourceC using a **LOAD FROM CURSOR** command:

```
DECLARE mycurs CURSOR FOR SELECT * FROM source
LOAD FROM mycurs OF CURSOR REPLACE INTO sourceC
```

If this command fails because the original data is incompatible with the definition of table sourceC, you must transform the data in the original table as it is being transferred to sourceC.

4. After the data is successfully copied to sourceC, submit the ALTER TABLE *target* ...ATTACH *sourceC* statement.

## Declaring temporary tables

To define temporary tables from within your applications, use the DECLARE GLOBAL TEMPORARY TABLE statement.

### About this task

Temporary tables, also referred to as user-defined temporary tables, are used by applications that work with data in the database. Results from manipulation of the data need to be stored temporarily in a table. A user temporary table space must exist before declaring temporary tables.

**Note:** The description of temporary tables does not appear in the system catalog thus making it not persistent for, and not able to be shared with, other applications. When the application using this table terminates or disconnects from the database, any data in the table is deleted and the table is implicitly dropped. Temporary tables do not support:

- User-defined type columns
- LONG VARCHAR columns
- XML columns for created global temporary tables

### Example

```
DECLARE GLOBAL TEMPORARY TABLE temptbl
   LIKE empltabl
   ON COMMIT DELETE ROWS
   NOT LOGGED
   IN usr_tbsp
```

This statement defines a temporary table called `temptbl`. This table is defined with columns that have exactly the same name and description as the columns of the `empltabl`. The implicit definition only includes the column name, data type, nullability characteristic, and column default value attributes. All other column attributes including unique constraints, foreign key constraints, triggers, and indexes are not defined. With ON COMMIT DELETE ROWS (any DELETE ROWS option), the database manager always deletes rows whether there's a cursor with a HOLD open on the table or not. The database manager optimizes a NOT LOGGED delete by implementing an internal TRUNCATE, if no WITH HOLD cursors are open, otherwise, the database manager deletes the rows one at a time.

The table is dropped implicitly when the application disconnects from the database. For more information, see the DECLARE GLOBAL TEMPORARY TABLE statement.

## Creating and connecting to created temporary tables

*Created temporary tables* are created using the CREATE GLOBAL TEMPORARY TABLE statement. The first time an application refers to a created temporary table

using a connection, a private version of the created temporary table is instantiated for use by the application using the connection.

## About this task

Similar to declared temporary tables, created temporary tables are used by applications that work with data in the database, where the results from manipulation of the data need to be stored temporarily in a table. Whereas declared temporary table information *is not* saved in the system catalog tables, and must be defined in every session where it is used, created temporary table information *is* saved in the system catalog and is not required to be defined in every session where it is used, thus making it persistent and able to be shared with other applications, across different connections. A user temporary table space must exist before created temporary tables can be created.

**Note:** The first implicit or explicit reference to the created temporary table that is executed by any program using the connection creates an empty instance of the given created temporary table. Each connection that references this created temporary table has its own unique instance of the created temporary table, and the instance is not persistent beyond the life of the connection.

References to the created temporary table name in multiple connections refer to the same, single, persistent created temporary table definition, and to a distinct instance of the created temporary table for each connection at the current server. If the created temporary table name that is being referenced is not qualified, it is implicitly qualified using the standard qualification rules that apply to SQL statements.

The owner implicitly has all table privileges on the created temporary table, including the authority to drop it. The owner's table privileges can be granted and revoked, either individually or with the ALL clause. Another authorization ID can access the created temporary table only if it has been granted appropriate privileges.

Indexes and SQL statements that modify data (such as INSERT, UPDATE, and DELETE) are supported. Indexes can only be created in the same table space as the created temporary table.

For the CREATE GLOBAL TEMPORARY TABLE statement: locking and recovery do not apply; logging applies only when the LOGGED clause is specified. For more options, see the CREATE GLOBAL TEMPORARY statement.

Created temporary tables cannot be:

- Associated with security policies
- Table partitioned
- Multidimensional clustering (MDC) tables
- Insert time clustering (ITC) tables
- Range-clustered (RCT)
- Distributed by replication

Materialized query tables (MQTs) cannot be created on created temporary tables.

Created temporary tables do not support the following column types, object types, and table or index operations:

- XML columns
- Structured types
- Referenced types
- Constraints
- Index extensions
- LOAD
- LOAD TABLE
- ALTER TABLE
- RENAME TABLE
- RENAME INDEX
- REORG TABLE
- REORG INDEX
- LOCK TABLE

For more information, see the CREATE GLOBAL TEMPORARY TABLE statement.

### Example

```
CREATE GLOBAL TEMPORARY TABLE temptbl
   LIKE empltabl
   ON COMMIT DELETE ROWS
   NOT LOGGED
   IN usr_tbsp
```

This statement creates a temporary table called `temptbl`. This table is defined with columns that have exactly the same name and description as the columns of the `empltabl`. The implicit definition only includes the column name, data type, nullability characteristic, and column default value attributes of the columns in `empltabl`. All other column attributes including unique constraints, foreign key constraints, triggers, and indexes are not implicitly defined.

A COMMIT always deletes the rows from the table. If there are any HOLD cursors open on the table, they can be deleted using TRUNCATE statement, which is faster, but will “normally” have to be deleted row by row. Changes made to the temporary table are not logged. The temporary table is placed in the specified user temporary table space, `usr tbsp`. This table space must exist or the creation of this table will fail.

When an application that instantiated a created temporary table disconnects from the database, the application's instance of the created temporary table is dropped.

## Distinctions between DB2 base tables and temporary tables

DB2 base tables and the two types of temporary tables have several distinctions.

The following table summarizes important distinctions between base tables, created temporary tables, and declared temporary tables.

*Table 13. Important distinctions between DB2 base tables and DB2 temporary tables*

| Area of distinction | Distinction |
|---|---|
| Creation, persistence, and ability to share table descriptions | **Base tables:** The CREATE TABLE statement puts a description of the table in the catalog view SYSCAT.TABLES. The table description is persistent and is shareable across different connections. The name of the table in the CREATE TABLE statement can be qualified. If the table name is not qualified, it is implicitly qualified using the standard qualification rules applied to SQL statements. |
| | **Created temporary tables:** The CREATE GLOBAL TEMPORARY TABLE statement puts a description of the table in the catalog view SYSCAT.TABLES. The table description is persistent and is shareable across different connections. The name of the table in the CREATE GLOBAL TEMPORARY TABLE statement can be qualified. If the table name is not qualified, it is implicitly qualified using the standard qualification rules applied to SQL statements. |
| | **Declared temporary tables:** The DECLARE GLOBAL TEMPORARY TABLE statement does not put a description of the table in the catalog. The table description is not persistent beyond the life of the connection that issued the DECLARE GLOBAL TEMPORARY TABLE statement and the description is known only to that connection.<br><br>Thus, each connection could have its own possibly unique description of the same declared temporary table. The name of the table in the DECLARE GLOBAL TEMPORARY TABLE statement can be qualified. If the table name is qualified, SESSION must be used as the schema qualifier. If the table name is not qualified, SESSION is implicitly used as the qualifier. |
| Table instantiation and ability to share data | **Base tables:** The CREATE TABLE statement creates one empty instance of the table, and all connections use that one instance of the table. The table and data are persistent. |
| | **Created temporary tables:** The CREATE GLOBAL TEMPORARY TABLE statement does not create an instance of the table. The first implicit or explicit reference to the table in an open, select, insert, update, or delete operation that is executed by any program using the connection creates an empty instance of the given table. Each connection that references the table has its own unique instance of the table, and the instance is not persistent beyond the life of the connection. |
| | **Declared temporary tables:** The DECLARE GLOBAL TEMPORARY TABLE statement creates an empty instance of the table for the connection. Each connection that declares the table has its own unique instance of the table, and the instance is not persistent beyond the life of the connection. |

*Table 13. Important distinctions between DB2 base tables and DB2 temporary tables (continued)*

| Area of distinction | Distinction |
|---|---|
| References to the table during the connection | **Base tables:** References to the table name in multiple connections refer to the same single persistent table description and to the same instance at the current server. If the table name that is being referenced is not qualified, it is implicitly qualified using the standard qualification rules that apply to SQL statements. |
| | **Created temporary tables:** References to the table name in multiple connections refer to the same single persistent table description but to a distinct instance of the table for each connection at the current server. If the table name that is being referenced is not qualified, it is implicitly qualified using the standard qualification rules that apply to SQL statements. |
| | **Declared temporary tables:** References to the table name in multiple connections refer to a distinct description and instance of the table for each connection at the current server. References to the table name in an SQL statement (other than the DECLARE GLOBAL TEMPORARY TABLE statement) must include SESSION as the schema qualifier. If the table name is not qualified with SESSION, the reference is assumed to be to a base table. |
| Table privileges and authorization | **Base tables:** The owner implicitly has all table privileges on the table and the authority to drop the table. The owner's table privileges can be granted and revoked, either individually or with the ALL clause.<br><br>Another authorization ID can access the table only if it has been granted appropriate privileges for the table. |
| | **Created temporary tables:** The owner implicitly has all table privileges on the table and the authority to drop the table. The owner's table privileges can be granted and revoked, either individually or with the ALL clause.<br><br>Another authorization ID can access the table only if it has been granted appropriate privileges for the table. |
| | **Declared temporary tables:** PUBLIC implicitly has all table privileges on the table without GRANT authority and also has the authority to drop the table. These table privileges cannot be granted or revoked.<br><br>Any authorization ID can access the table without requiring a grant of any privileges for the table. |
| Indexes and other SQL statement support | **Base tables:** Indexes and SQL statements that modify data (INSERT, UPDATE, DELETE, and so on) are supported. Indexes can be in different table spaces. |
| | **Created temporary tables:** Indexes and SQL statements that modify data (INSERT, UPDATE, DELETE, and so on) are supported. Indexes can only be in the same table space as the table. |
| | **Declared temporary tables:** Indexes and SQL statements that modify data (INSERT, UPDATE, DELETE, and so on) are supported. Indexes can only be in the same table space as the table. |

*Table 13. Important distinctions between DB2 base tables and DB2 temporary tables (continued)*

| Area of distinction | Distinction |
|---|---|
| Locking, logging, and recovery | **Base tables:** Locking, logging, and recovery do apply. |
| | **Created temporary tables:** Locking and recovery do not apply, however logging does apply when LOGGED is explicitly specified. Undo recovery (rolling back changes to a savepoint or the most recent commit point) is supported when only when LOGGED is *explicitly* specified. |
| | **Declared temporary tables:** Locking and recovery do not apply, however logging only applies when LOGGED is explicitly or implicitly specified. Undo recovery (rolling back changes to a savepoint or the most recent commit point) is supported when LOGGED is explicitly or implicitly specified. |

## Creating tables with XML columns

To create tables with XML columns, you specify columns with the XML data type in the CREATE TABLE statement. A table can have one or more XML columns.

You do not specify a length when you define an XML column. However, serialized XML data that is exchanged with a DB2 database is limited to 2 GB per value of type XML, so the effective limit of an XML document is 2 GB.

Like a LOB column, an XML column holds only a descriptor of the column. The data is stored separately.

**Note:**

- If you enable data row compression for the table, XML documents require less storage space.
- You can optionally store smaller and medium-size XML documents in the row of the base table instead of storing them in the default XML storage object.

**Example:** The sample database contains a table for customer data that contains two XML columns. The definition looks like this:

```
CREATE TABLE Customer (Cid BIGINT NOT NULL PRIMARY KEY,
                       Info XML,
                       History XML)
```

**Example:** The VALIDATED predicate checks whether the value in the specified XML column has been validated. You can define a table check constraint on XML columns, using the VALIDATED predicate, to ensure that all documents inserted or updated in a table are valid.

```
CREATE TABLE TableValid (id BIGINT,
                         xmlcol XML,
                         CONSTRAINT valid_check CHECK (xmlcol IS VALIDATED))
```

**Example:** Setting the COMPRESS attribute to YES enables data row compression. XML documents stored in XML columns are subject to row compression. Compressing data at the row level allows repeating patterns to be replaced with shorter symbol strings.

```
CREATE TABLE TableXmlCol (id BIGINT,
                          xmlcol XML) COMPRESS YES
```

**Example:** The following CREATE TABLE statement creates a patient table partitioned by visit date. All records between January 01, 2000 and December 31, 2006 are in the first partition. The more recent data are partitioned every 6 months.

```
CREATE TABLE Patients ( patientID BIGINT, visit_date DATE, diagInfo XML,
    prescription XML )
    INDEX IN indexTbsp LONG IN ltbsp
    PARTITION BY ( visit_date )
       ( STARTING '1/1/2000' ENDING '12/31/2006',
         STARTING '1/1/2007' ENDING '6/30/2007',
            ENDING '12/31/2007',
            ENDING '6/30/2008',
            ENDING '12/31/2008',
            ENDING '6/30/2009' );
```

## Adding XML columns to existing tables

To add XML columns to existing tables, you specify columns with the XML data type in the ALTER TABLE statement with the ADD clause. A table can have one or more XML columns.

**Example** The sample database contains a table for customer data that contains two XML columns. The definition looks like this:

```
CREATE TABLE Customer (Cid BIGINT NOT NULL PRIMARY KEY,
                       Info XML,
                       History XML)
```

Create a table named MyCustomer that is a copy of Customer, and add an XML column to describe customer preferences:

```
CREATE TABLE MyCustomer LIKE Customer;
ALTER TABLE MyCustomer ADD COLUMN Preferences XML;
```

**Example:** Setting the COMPRESS attribute to YES enables data row compression. XML documents stored in XML columns are subject to row compression. Compressing data at the row level allows repeating patterns to be replaced with shorter symbol strings.

```
ALTER TABLE MyCustomer ADD COLUMN Preferences XML COMPRESS YES;
```

**Example:** The following CREATE TABLE statement creates a patient table partitioned by visit date. All records between January 01, 2000 and December 31, 2006 are in the first partition. The more recent data are partitioned every 6 months.

```
CREATE TABLE Patients ( patientID INT, Name Varchar(20), visit_date DATE,
      diagInfo XML )
    PARTITION BY ( visit_date )
       ( STARTING '1/1/2000' ENDING '12/31/2006',
         STARTING '1/1/2007' ENDING '6/30/2007',
            ENDING '12/31/2007',
            ENDING '6/30/2008',
            ENDING '12/31/2008',
            ENDING '6/30/2009' );
```

The following ALTER table statement adds another XML column for patient prescription information:

```
ALTER TABLE Patients ADD COLUMN prescription XML ;
```

## Creating partitioned tables

Partitioned tables use a data organization scheme in which table data is divided across multiple storage objects, called data partitions or ranges, according to values in one or more table partitioning key columns of the table. Data from a given table is partitioned into multiple storage objects based on the specifications provided in the PARTITION BY clause of the CREATE TABLE statement. These storage objects can be in different table spaces, in the same table space, or a combination of both.

### Before you begin

To create a table, the privileges held by the authorization ID of the statement must include at least one of the following authorities or privileges:

- CREATETAB authority on the database and USE privilege on all the table spaces used by the table, as well as one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the table does not exist
  - CREATEIN privilege on the schema, if the schema name of the table refers to an existing schema
- DBADM authority

### About this task

You can create a partitioned table by using the CREATE TABLE statement.

### Procedure

To create a partitioned table from the command line, issue the CREATE TABLE statement:

```
CREATE TABLE NAME (column_name data_type null_attribute) IN
 table_space_list PARTITION BY RANGE (column_expression)
 STARTING FROM constant ENDING constant EVERY constant
```

For example, the following statement creates a table where rows with a ≥ 1 and a ≤ 20 are in PART0 (the first data partition), rows with 21 ≤ a ≤ 40 are in PART1 (the second data partition), up to 81 ≤ a ≤ 100 are in PART4 (the last data partition).

```
CREATE TABLE foo(a INT)
 PARTITION BY RANGE (a) (STARTING FROM (1)
 ENDING AT (100) EVERY (20))
```

## Defining ranges on partitioned tables

You can specify a range for each data partition when you create a partitioned table. A partitioned table uses a data organization scheme in which table data is divided across multiple data partitions according to the values of the table partitioning key columns of the table.

### About this task

Data from a given table is partitioned into multiple storage objects based on the specifications provided in the PARTITION BY clause of the CREATE TABLE statement. A range is specified by the STARTING FROM and ENDING AT values of the PARTITION BY clause.

To completely define the range for each data partition, you must specify sufficient boundaries. The following is a list of guidelines to consider when defining ranges on a partitioned table:

- The STARTING clause specifies a low boundary for the data partition range. This clause is mandatory for the lowest data partition range (although you can define the boundary as MINVALUE). The lowest data partition range is the data partition with the lowest specified bound.
- The ENDING (or VALUES) clause specifies a high boundary for the data partition range. This clause is mandatory for the highest data partition range (although you can define the boundary as MAXVALUE). The highest data partition range is the data partition with the highest specified bound.
- If you do not specify an ENDING clause for a data partition, then the next greater data partition must specify a STARTING clause. Likewise, if you do not specify a STARTING clause, then the previous data partition must specify an ENDING clause.
- MINVALUE specifies a value that is smaller than any possible value for the column type being used. MINVALUE and INCLUSIVE or EXCLUSIVE cannot be specified together.
- MAXVALUE specifies a value that is larger than any possible value for the column type being used. MAXVALUE and INCLUSIVE or EXCLUSIVE cannot be specified together.
- INCLUSIVE indicates that all values equal to the specified value are to be included in the data partition containing this boundary.
- EXCLUSIVE indicates that all values equal to the specified value are NOT to be included in the data partition containing this boundary.
- The NULL clause of the CREATE TABLE statement specifies whether null values are to be sorted high or low when considering data partition placement. By default, null values are sorted high. Null values in the table partitioning key columns are treated as positive infinity, and are placed in a range ending at MAXVALUE. If no such data partition is defined, null values are considered to be out-of-range values. Use the NOT NULL constraint if you want to exclude null values from table partitioning key columns. LAST specifies that null values are to appear last in a sorted list of values. FIRST specifies that null values are to appear first in a sorted list of values.
- When using the long form of the syntax, each data partition must have at least one bound specified.

**Tip:** Before you begin defining data partitions on a table it is important to understand how tables benefit from table partitioning and what factors influence the columns you choose as partitioning columns.

The ranges specified for each data partition can be generated automatically or manually.

**Automatically generated**

Automatic generation is a simple method of creating many data partitions quickly and easily. This method is appropriate for equal sized ranges based on dates or numbers.

Examples 1 and 2 demonstrate how to use the CREATE TABLE statement to define and generate automatically the ranges specified for each data partition.

Example 1:

Issue a create table statement with the following ranges defined:

```
CREATE TABLE lineitem (
  l_orderkey    DECIMAL(10,0) NOT NULL,
  l_quantity    DECIMAL(12,2),
  l_shipdate    DATE,
  l_year_month  INT GENERATED ALWAYS AS (YEAR(l_shipdate)*100 + MONTH(l_shipdate)))
     PARTITION BY RANGE(l_shipdate)
    (STARTING ('1/1/1992') ENDING ('12/31/1992') EVERY 1 MONTH);
```

This statement results in 12 data partitions each with 1 key value (l_shipdate) >= ('1/1/1992'), (l_shipdate) < ('3/1/1992'), (l_shipdate) < ('4/1/1992'), (l_shipdate) < ('5/1/1992'), ..., (l_shipdate) < ('12/1/1992'), (l_shipdate) < ('12/31/1992').

The starting value of the first data partition is inclusive because the overall starting bound ('1/1/1992') is inclusive (default). Similarly, the ending bound of the last data partition is inclusive because the overall ending bound ('12/31/1992') is inclusive (default). The remaining STARTING values are inclusive and the remaining ENDING values are all exclusive. Each data partition holds n key values where n is given by the EVERY clause. Use the formula (start + every) to find the end of the range for each data partition. The last data partition might have fewer key values if the EVERY value does not divide evenly into the START and END range.

Example 2:

Issue a create table statement with the following ranges defined:

```
CREATE TABLE t(a INT, b INT)
 PARTITION BY RANGE(b) (STARTING FROM (1)
  EXCLUSIVE ENDING AT (1000) EVERY (100))
```

This statement results in 10 data partitions each with 100 key values (1 < b <= 101, 101 < b <= 201, ..., 901 < b <= 1000).

The starting value of the first data partition (b > 1 and b <= 101) is exclusive because the overall starting bound (1) is exclusive. Similarly the ending bound of the last data partition ( b > 901 b <= 1000) is inclusive because the overall ending bound (1000) is inclusive. The remaining STARTING values are all exclusive and the remaining ENDING values are all inclusive. Each data partition holds n key values where n is given by the EVERY clause. Finally, if both the starting and ending bound of the overall clause are exclusive, the starting value of the first data partition is exclusive because the overall starting bound (1) is exclusive. Similarly the ending bound of the last data partition is exclusive because the overall ending bound (1000) is exclusive. The remaining STARTING values are all exclusive and the ENDING values are all inclusive. Each data partition (except the last) holds n key values where n is given by the EVERY clause.

**Manually generated**

Manual generation creates a new data partition for each range listed in the PARTITION BY clause. This form of the syntax allows for greater flexibility when defining ranges thereby increasing your data and LOB placement options. Examples 3 and 4 demonstrate how to use the CREATE TABLE statement to define and generate manually the ranges specified for a data partition.

Example 3:

This statement partitions on two date columns both of which are generated. Notice the use of the automatically generated form of the CREATE TABLE syntax and that only one end of each range is specified. The other end is implied from the adjacent data partition and the use of the INCLUSIVE option:

```
CREATE TABLE sales(invoice_date date, inv_month int NOT NULL
GENERATED ALWAYS AS (month(invoice_date)), inv_year INT NOT
NULL GENERATED ALWAYS AS ( year(invoice_date)),
item_id int NOT NULL,
cust_id int NOT NULL) PARTITION BY RANGE (inv_year,
inv_month)
(PART Q1_02 STARTING (2002,1) ENDING (2002, 3) INCLUSIVE,
PART Q2_02 ENDING (2002, 6) INCLUSIVE,
PART Q3_02 ENDING (2002, 9) INCLUSIVE,
PART Q4_02 ENDING (2002,12) INCLUSIVE,
PART CURRENT ENDING (MAXVALUE, MAXVALUE));
```

Gaps in the ranges are permitted. The CREATE TABLE syntax supports gaps by allowing you to specify a STARTING value for a range that does not line up against the ENDING value of the previous data partition.

Example 4:

Creates a table with a gap between values 101 and 200.

```
CREATE TABLE foo(a INT)
 PARTITION BY RANGE(a)
   (STARTING FROM (1) ENDING AT (100),
   STARTING FROM (201) ENDING AT (300))
```

Use of the ALTER TABLE statement, which allows data partitions to be added or removed, can also cause gaps in the ranges.

When you insert a row into a partitioned table, it is automatically placed into the proper data partition based on its key value and the range it falls within. If it falls outside of any ranges defined for the table, the insert fails and the following error is returned to the application:

```
SQL0327N  The row cannot be inserted into table <tablename>
because it is outside the bounds of the defined data partition ranges.
   SQLSTATE=22525
```

**Restrictions**

- Table level restrictions:
  - Tables created using the automatically generated form of the syntax (containing the EVERY clause) are constrained to use a numeric or date time type in the table partitioning key.
- Statement level restrictions:
  - MINVALUE and MAXVALUE are not supported in the automatically generated form of the syntax.
  - Ranges are ascending.
  - Only one column can be specified in the automatically generated form of the syntax.
  - The increment in the EVERY clause must be greater than zero.
  - The ENDING value must be greater than or equal to the STARTING value.

## Renaming tables and columns

You can use the RENAME statement to rename an existing table. To rename columns, use the ALTER TABLE statement.

### About this task

When renaming tables, the source table must not be referenced in any existing definitions (view or materialized query table), triggers, SQL functions, or constraints. It must also not have any generated columns (other than identity columns), or be a parent or dependent table. Catalog entries are updated to reflect the new table name. For more information and examples, see the RENAME statement.

The RENAME COLUMN clause is an option on the ALTER TABLE statement. You can rename an existing column in a base table to a new name without losing stored data or affecting any privileges or label-based access control (LBAC) policies that are associated with the table.

Only the renaming of base table columns is supported. Renaming columns in views, materialized query tables (MQTs), declared and created temporary tables, and other table-like objects is not supported.

Invalidation and revalidation semantics for the rename column operation are similar to those for the drop column operation; that is, all dependent objects are invalidated. Revalidation of all dependent objects following a rename column operation is always done immediately after the invalidation, even if the **auto_reval** database configuration parameter is set to DISABLED.

The following example shows the renaming of a column using the ALTER TABLE statement:

```
ALTER TABLE org RENAME COLUMN deptnumb TO deptnum
```

To change the definition of existing columns, see the "Changing column properties" topic or the ALTER TABLE statement.

## Viewing table definitions

You can use the SYSCAT.TABLES and SYSCAT.COLUMNS catalog views to view table definitions. For SYSCAT.COLUMNS, each row represents a column defined for a table, view, or nickname. To see the data in the columns, use the SELECT statement.

### About this task

You can also use the following views and table functions to view table definitions:

- ADMINTEMPCOLUMNS administrative view
- ADMINTEMPTABLES administrative view
- ADMIN_GET_TEMP_COLUMNS table function - Retrieve column information for temporary tables
- ADMIN_GET_TEMP_TABLES table function - Retrieve information for temporary tables

# Dropping application-period temporal tables

Use the DROP TABLE statement to drop application-period temporal tables.

## About this task

When a table is dropped, the row in the SYSCAT.TABLES system catalog view that contains information about that table is dropped, and any other objects that depend on the table are affected. For example:

- All column names are dropped.
- Indexes created on any columns of the table are dropped.
- All views based on the table are marked inoperative.
- All privileges on the dropped table and dependent views are implicitly revoked.
- All referential constraints in which the table is a parent or dependent are dropped.
- All packages and cached dynamic SQL and XQuery statements dependent on the dropped table are marked invalid, and remain so until the dependent objects are re-created. This includes packages dependent on any supertable above the subtable in the hierarchy that is being dropped.
- Any reference columns for which the dropped table is defined as the scope of the reference become “unscoped”.
- An alias definition on the table is not affected, because an alias can be undefined
- All triggers dependent on the dropped table are marked inoperative.

Restrictions

An individual table cannot be dropped if it has a subtable.

## Procedure

- To drop a table, use a DROP TABLE statement.

  The following statement drops the table called DEPARTMENT:

  ```
  DROP TABLE DEPARTMENT
  ```

- To drop all the tables in a table hierarchy, use a DROP TABLE HIERARCHY statement.

  The DROP TABLE HIERARCHY statement must name the root table of the hierarchy to be dropped. For example:

  ```
  DROP TABLE HIERARCHY person
  ```

## Results

There are differences when dropping a table hierarchy compared to dropping a specific table:

- DROP TABLE HIERARCHY does not activate deletion-triggers that would be activated by individual DROP TABLE statements. For example, dropping an individual subtable would activate deletion-triggers on its supertables.
- DROP TABLE HIERARCHY does not make log entries for the individual rows of the dropped tables. Instead, the dropping of the hierarchy is logged as a single event.

# Chapter 20. Temporal tables

You can use temporal tables to associate time-based state information with your data. Data in tables that do not use temporal support are deemed to be applicable to the present, while data in temporal tables can be valid for a period defined by the database system, user applications, or both.

There are many business needs requiring the storage and maintenance of time-based data. Without this capability in a database, it is expensive and complex to maintain a time-focused data support infrastructure. With temporal tables, the database can store and retrieve time-based data without additional application logic. For example, a database can store the history of a table (deleted rows or the original values of rows that have been updated) so you can query the past state of your data. You can also assign a date range to a row of data to indicate when it is deemed to be valid by your applications or business rules.

A temporal table records the period when a row is valid. A period is an interval of time that is defined by two date or time columns in the temporal table. A period contains a begin column and an end column. The begin column indicates the beginning of the period, and the end column indicates the end of the period. The beginning value of a period is inclusive, while the ending value of a period is exclusive. For example, a row with a period from January 1 to February 1 is valid from January 1, until January 31 at midnight.

Two period types are supported:

**System periods**
A system period consists of a pair of columns with database manager-maintained values that indicate the period when a row is current. The begin column contains a timestamp value for when a row was created. The end column contains a timestamp value for when a row was updated or deleted. When a system-period temporal table is created, it contains the currently active rows. Each system-period temporal table is associated with a history table that contains any changed rows.

**Application periods**
An application period consists of a pair of columns with user or application-supplied values that indicate the period when a row is valid. The begin column indicates the time when a row is valid from. The end column indicates the time when a row stops being valid. A table with an application period is called an application-period temporal table.

You can check whether a table has temporal support by querying the SYSCAT.TABLES system catalog view. For example:

```
SELECT TABSCHEMA, TABNAME, TEMPORALTYPE FROM SYSCAT.TABLES
```

The returned values for `TEMPORALTYPE` are defined as follows:

**A** Application-period temporal table

**B** Bitemporal table

**N** Not a temporal table

**S** System-period temporal table

## System-period temporal tables

A system-period temporal table is a table that maintains historical versions of its rows. Use a system-period temporal table to store current versions of your data and use its associated history table to transparently store your updated and deleted data rows.

A system-period temporal table includes a SYSTEM_TIME period with columns that capture the begin and end times when the data in a row is current. The database manager also uses the SYSTEM_TIME period to preserve historical versions of each table row whenever updates or deletes occur. The database manager stores these rows in a history table that is exclusively associated with a system-period temporal table. Adding versioning establishes the link between the system-period temporal table and the history table. With a system-period temporal table, your queries have access to your data at the current point in time and the ability to retrieve data from past points in time.

A system-period temporal table also includes a transaction start-ID column. This column captures the time when execution started for a transaction that impacts the row. If multiple rows are inserted or updated within a single SQL transaction, then the values for the transaction start-ID column are the same for all the rows and are unique from the values generated for this column by other transactions. This common start-ID column value means you can use the transaction start-ID column to identify all the rows in the tables that were written by the same transaction.

### History tables

Each system-period temporal table requires a history table. When a row is updated or deleted from a system-period temporal table, the database manager inserts a copy of the old row into its associated history table. This storage of old system-period temporal table data gives you the ability to retrieve data from past points in time.

In order to store row data, the history table columns and system-period temporal table columns must have the same names, order, and data types. You can create a history table with the same names and descriptions as the columns of the system-period temporal table by using the LIKE clause of the CREATE TABLE statement, for example:

```
CREATE TABLE employees_history LIKE employees IN hist_space;
```

An existing table can be used as a history table if it avoids the restrictions listed in the description of the ALTER TABLE statement USE HISTORY clause.

After you create a history table, you add versioning to establish the link between the system-period temporal table and the history table.

```
ALTER TABLE employees ADD VERSIONING USE HISTORY TABLE employees_history;
```

A history table is subject to the following rules and restrictions when versioning is enabled:

- A history table cannot explicitly be dropped. It can only implicitly be dropped when the associated system-period temporal table is dropped.
- History table columns cannot explicitly be added, dropped, or changed.
- A history table must not be defined as parent, child, or self-referencing in a referential constraint. Access to the history table is restricted to prevent cascaded actions to the history table.

- A table space that contains a history table, but not its associated system-period temporal table, cannot be dropped.

You should rarely need to explicitly change a history table. Doing so might jeopardize your ability to audit a system-period temporal table data history. You should restrict access to a history table to protect its data.

Under normal operations, a history table experiences mostly insert and read activities. Updates and deletes are rare. The absence of updates and deletes means that history tables typically do not have free space that can be reused for the inserting of new rows. If row inserts into the history table are negatively impacting workload performance, you can eliminate the search for free space by altering the definition of the history table by using the APPEND ON option. This option avoids the processing associated with free space searches and directly appends new rows to the end of the table.

```
ALTER TABLE employees_history APPEND ON;
```

When a system-period temporal table is dropped, the associated history table and any indexes defined on the history table are implicitly dropped. To avoid losing historical data when a system-period temporal table is dropped, you can either create the history table with the RESTRICT ON DROP attribute or alter the history table by adding the RESTRICT ON DROP attribute.

```
CREATE TABLE employees_history LIKE employees WITH RESTRICT ON DROP;
```

Because history tables experience more inserts than deletes, your history tables are always growing and so are consuming an increasing amount of storage. Deciding how to prune your history tables to get rid of the rows that you no longer need can be a complex task. You need to understand the value of your individual records. Some content, like customer contracts, might be untouchable and can never be deleted. While other records, like website visitor information, can be pruned without concern. Often it is not the age of a row that determines when it can be pruned and archived, but rather it is some business logic that is the deciding factor. The following list contains some possible rules for pruning:

- Prune rows selected by a user-supplied query that reflects business rules.
- Prune rows older than a certain age.
- Prune history rows when more than N versions exist for that record (retain only the latest N versions).
- Prune history rows when the record is deleted from the associated system-period temporal table (when there are no current versions).

There are several ways to periodically prune old data from a history table:

- Use range partitioning and detach old partitions from the history table.
- Use DELETE statements to remove rows from the table. If using DELETE statements, you might observe the following guidelines:
  - Periodically reorganize the history table to release the free space left behind by the delete operations.
  - Ensure that the history table was not altered to use the APPEND ON option, allowing inserts to search for free space.

## SYSTEM_TIME period

The SYSTEM_TIME period columns for a system-period temporal table indicate when the version of a row is current.

The SYSTEM_TIME period contains a pair of TIMESTAMP(12) columns whose values are generated by the database manager. The columns must be defined as NOT NULL with an applicable GENERATED ALWAYS AS option. The begin column of the period must be a row-begin column and the end column of the period must be a row-end column.

```
CREATE TABLE policy_info
(
 policy_id       CHAR(4) NOT NULL,
 coverage        INT NOT NULL,
 sys_start   TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,
 sys_end  TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,
 ts_id   TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS TRANSACTION START ID,
 PERIOD SYSTEM_TIME (sys_start, sys_end)
) IN policy_space;
```

**Row-begin column**

This column represents the time when the row data became current. The database manager generates a value for this column by using a reading of the system clock at the moment it executes the first data change statement in the transaction that generates the row. If multiple rows are inserted or updated within a single SQL transaction, the values for the row-begin column are the same for all the impacted rows. The values for these row-begin columns are unique from the values generated for the row-begin columns for other transactions. A row-begin column is required as the begin column of a SYSTEM_TIME period, which must be defined for each system-period temporal table.

When an existing regular table is altered to make it a system-period temporal table, a row-begin column is added to the table. The row-begin column is populated with a default of `0001-01-01-00.00.00.000000000000` for the TIMESTAMP(12) data type value for any existing rows.

**Row-end column**

This column represents the time when the row data was no longer current. For rows in a history table, the value in the row-end column represents when the row was added to the history table. The rows in the system-period temporal table are by definition current, so the row-end column is populated with a default value for the TIMESTAMP(12) data type (for example: `9999-12-30-00.00.00.000000000000`). A row-end column is required as the end column of a SYSTEM_TIME period, which must be defined for each system-period temporal table.

When an existing regular table is altered to make it a system-period temporal table, a row-end column is added to the table. The row-end column is populated with the maximum value for the TIMESTAMP(12) data type (default value: `9999-12-30-00.00.00.000000000000`) for any existing rows.

Since row-begin and row-end are generated columns, there is no implicit check constraint generated for SYSTEM_TIME that ensures that the value for an end column is greater than the value for its begin column in a system-period temporal table. This lack of a check constraint differs from an application-period temporal table where there is a check constraint associated with its BUSINESS_TIME. A row where the value for the end column is less than the value for the begin column cannot be returned when a period-specification is used to query the table. You can define a constraint to guarantee that the value for end column is greater than the value for begin column. This guarantee is useful when supporting operations that explicitly input data into these generated columns, such as a load operation.

The **systime_period_adj** database configuration parameter is used to specify what action to take when a history row for a system-period temporal table is generated with an end column value that is less than the value for begin column.

## Creating a system-period temporal table

Creating a system-period temporal table results in a table that tracks when data changes occur and preserves historical versions of that data.

### About this task

When creating a system-period temporal table, include attributes that indicate when data in a row is current and when transactions affected the data:

- Include row-begin and row-end columns that are used by the SYSTEM_TIME period to track when a row is current.
- Include a transaction start-ID column that captures the start times for transactions that affect rows.
- Create a history table to receive old rows from the system-period temporal table.
- Add versioning to establish the link between the system-period temporal table and the history table.

The row-begin, row-end, and transaction start-ID columns can be defined as IMPLICITLY HIDDEN. Since these columns and their entries are generated by the database manager, hiding them can minimize any potential negative affects on your applications. These columns are then unavailable unless referenced, for example:

- A SELECT * query run against a table does not return any implicitly hidden columns in the result table.
- An INSERT statement does not expect a value for any implicitly hidden columns.
- The LOAD, IMPORT, and EXPORT commands can use the *includeimplicitlyhidden* modifier to work with implicitly hidden columns.

A system-period temporal table can be defined as a parent or a child in a referential constraint. However, the referential constraints are applied only to the current data, that is the data in the system-period temporal table. The constraints are not applied to the associated history table. In order to minimize inconsistencies when a system-period temporal table is a child table in a referential constraint, the parent table should also be a system-period temporal table.

**Note:** While the row-begin, row-end, and transaction start-ID generated columns are required when creating a system-period temporal table, you can also create a regular table with these generated columns.

The example in the following section shows the creation of a table that stores policy information for the customers of an insurance company.

### Procedure

To create a system-period temporal table.

1. Create a table with a SYSTEM_TIME attribute. For example:

```
CREATE TABLE policy_info
(
 policy_id    CHAR(4) NOT NULL,
 coverage     INT NOT NULL,
```

```
 sys_start    TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,
 sys_end      TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,
 ts_id        TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS TRANSACTION START ID,
 PERIOD SYSTEM_TIME (sys_start, sys_end)
) IN policy_space;
```

2. Create a history table. For example:

```
CREATE TABLE hist_policy_info
(
 policy_id    CHAR(4) NOT NULL,
 coverage     INT NOT NULL,
 sys_start    TIMESTAMP(12) NOT NULL,
 sys_end      TIMESTAMP(12) NOT NULL,
 ts_id        TIMESTAMP(12) NOT NULL
) IN hist_space;
```

   You can also create a history table with the same names and descriptions as the columns of the system-period temporal table by using the LIKE clause of the CREATE TABLE statement. For example:

```
CREATE TABLE hist_policy_info LIKE policy_info IN hist_space;
```

3. Add versioning to the system-period temporal table to establish a link to the history table. For example:

```
ALTER TABLE policy_info ADD VERSIONING USE HISTORY TABLE hist_policy_info;
```

## Results

The `policy_info` table stores the insurance coverage level for a customer. The SYSTEM_TIME period related columns (`sys_start` and `sys_end`) show when a coverage level row is current. The `ts_id` column lists the time when execution started for a transaction that impacted the row.

*Table 14. Created system-period temporal table (policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| | | | | |

The `hist_policy_info` history table receives the old rows from the `policy_info` table.

*Table 15. Created history table (hist_policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| | | | | |

## Example

This section contains more creating system-period temporal table examples.

**Hiding columns**

The following example creates the `policy_info` table with the TIMESTAMP(12) columns (`sys_start`, `sys_end` and `ts_id`) marked as implicitly hidden.

```
CREATE TABLE policy_info
(
 policy_id    CHAR(4) NOT NULL,
 coverage     INT NOT NULL,
 sys_start    TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN IMPLICITLY HIDDEN,
 sys_end      TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END IMPLICITLY HIDDEN,
 ts_id        TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN,
 PERIOD SYSTEM_TIME (sys_start, sys_end)
) in policy_space;
```

Creating the hist_policy_info history table using the LIKE clause of the CREATE TABLE statement results in the history table inheriting the implicitly hidden attribute from the policy_info table. If you do not use the LIKE clause when creating the history table, then any columns marked as hidden in the system-period temporal table must also be marked as hidden in the associated history table.

**Changing an existing table into a system-period temporal table**

The following example adds timestamp columns and a SYSTEM_TIME period to an existing table (employees) enabling system-period temporal table functions.

```
ALTER TABLE employees
   ADD COLUMN sys_start TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN;
ALTER TABLE employees
   ADD COLUMN sys_end TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END;
ALTER TABLE employees
   ADD COLUMN ts_id TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS TRANSACTION START ID;
ALTER TABLE employees ADD PERIOD SYSTEM_TIME(sys_start, sys_end);
```

These new columns can be hidden by including the IMPLICITLY HIDDEN clause in the ALTER TABLE statement

A history table must be created and versioning added to finish this task.

## Dropping a system-period temporal table

Dropping a system-period temporal table also drops its associated history table and any indexes defined on the history table.

### Before you begin

To drop a system-period temporal table, you must be authorized to drop its history table.

### About this task

A history table is implicitly dropped when its associated system-period temporal table is dropped. A history table cannot be explicitly dropped by using the DROP statement.

To avoid losing historical data when a system-period temporal table is dropped, you can either create the history table with the RESTRICT ON DROP attribute or alter the history table by adding the RESTRICT ON DROP attribute. If you try to drop a system-period temporal table and its history table has the RESTRICT ON DROP attribute, the drop of the system-period temporal table fails (SQLSTATE 42893). In such cases, you must break the link between the system-period temporal table and the history table by removing the VERSIONING attribute and then rerun the DROP statement.

When a table is altered to drop VERSIONING, all packages with the versioning dependency on the table are invalidated. Other dependent objects, for example, views or triggers are marked invalid 'N' in the system catalog. Auto-revalidation is done. Any objects failing revalidation are left as invalid in the catalog. Some objects can become valid after only explicit user action.

## Procedure

To drop a system-period temporal table and its associated history table:

1. Optional: Protect historical data from deletion:
   a. If the history table was not created with the RESTRICT ON DROP attribute, alter the history table to set the RESTRICT ON DROP attribute. For example, if audit requirements made it necessary to preserve the history of insurance policies then the history table must be protected.

      ```
      ALTER TABLE hist_policy_info ADD RESTRICT ON DROP;
      ```

   b. Break the link between the system-period temporal table and a history table with RESTRICT ON DROP attribute by removing the VERSIONING attribute. For example:

      ```
      ALTER TABLE policy_info DROP VERSIONING;
      ```

2. Drop the system-period temporal table with the DROP statement. For example, the insurance policy tables created in the example in the Creating a system-period temporal table topic are no longer required.

   ```
   DROP TABLE policy_info;
   ```

## Results

The preceding commands affect the `policy_info` and `hist_policy_info` tables as follows:

- The DROP statement explicitly drops the system-period temporal table and implicitly drops the associated history table. The `policy_info` and `hist_policy_info` tables are deleted. Any objects that are directly or indirectly dependent on those tables are either deleted or made inoperative.
- After the RESTRICT ON DROP attribute is associated with the history table, any attempt to drop the `policy_info` table would fail (SQLSTATE 42893). A system-period temporal table can also be created or altered to use the RESTRICT ON DROP attribute.
- After the link between the system-period temporal table and its history table is broken, the `policy_info` table can be dropped and the `hist_policy_info` history table would remain.

### Dropping table spaces

If a table space contains a history table, but does not contain the associated system-period temporal table, that table space cannot be explicitly dropped. For example, using the insurance policy tables that were created in the `policy_space` and `hist_space` table spaces, the following statement is blocked:

```
DROP TABLESPACE hist_space;
```

If table space that contains a history table and the table space containing the associated system-period temporal table are included together, then the statement is allowed. For example, the following statement would succeed:

```
DROP TABLESPACE policy_space hist_space;
```

A history table is implicitly dropped when the table space for its associated system-period temporal table is dropped. For example, the following statement would drop the `hist_policy_info` history table:

```
DROP TABLESPACE policy_space;
```

## Application-period temporal tables

An application-period temporal table is a table that stores the in effect aspect of application data. Use an application-period temporal table to manage data based on time criteria by defining the time periods when data is valid.

Similar to a system-period temporal table, an application-period temporal table includes a BUSINESS_TIME period with columns that indicate the time period when the data in that row is valid or in effect. You provide the begin time and end time for the BUSINESS_TIME period associated with each row. However, unlike a system time-period temporal table, there is no separate history table. Past, present, and future effective dates and their associated business data are maintained in a single table. You can control data values by BUSINESS_TIME period and use application-period temporal tables for modeling data in the past, present, and future.

## BUSINESS_TIME period

The BUSINESS_TIME period columns for an application-period temporal table record when the version of a row is valid from a user or business application perspective.

The BUSINESS_TIME period contains a pair of DATE or TIMESTAMP(*p*) columns where *p* can be from 0 to 12. These columns are populated by you or a business application. The two columns in a BUSINESS_TIME period denote the start and end of the validity period. These columns differs from SYSTEM_TIME period columns where the time period values are generated by the database manager. The BUSINESS_TIME period columns must be defined as NOT NULL and must not be generated columns.

A BUSINESS_TIME period is inclusive-exclusive. The start of the validity period is included in the BUSINESS_TIME, while the end is excluded.

Whenever a BUSINESS_TIME period is defined on a table, an implicit check constraint named DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME is generated to ensure that the value for the end of the validity period is greater than the value for the start of the validity period. If a constraint with the same name exists, then an error is returned. This constraint is useful when supporting operations that explicitly input data into these columns, such as an insert or load operation.

An application-period temporal table can be defined so that rows with the same key do not have any overlapping periods of BUSINESS_TIME. For example, this restriction would prevent two versions of an insurance policy from being in effect at any point in time. These controls can be achieved by adding a BUSINESS_TIME WITHOUT OVERLAPS clause to a primary key, unique constraint specification, or create unique index statement. The enforcement is handled by the database manager. This control is optional.

## Creating an application-period temporal table

Creating an application-period temporal table results in a table that manages data based on when its data is valid or in effect.

## About this task

When creating an application-period temporal table, include a BUSINESS_TIME period that indicates when the data in a row is valid. You can optionally define that overlapping periods of BUSINESS_TIME are not allowed and that values are unique with respect to any period. The example in the following section shows the creation of a table that stores policy information for the customers of an insurance company.

## Procedure

To create an application-period temporal table:

1. Create a table with a BUSINESS_TIME period. For example:

```
CREATE TABLE policy_info
(
 policy_id     CHAR(4) NOT NULL,
 coverage      INT NOT NULL,
 bus_start     DATE NOT NULL,
 bus_end       DATE NOT NULL,
 PERIOD BUSINESS_TIME (bus_start, bus_end)
 );
```

2. Optional: Create a unique index that prevents overlapping periods of BUSINESS_TIME for the same `policy_id`. For example:

```
CREATE UNIQUE INDEX ix_policy
   ON policy_info (policy_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

## Results

The `policy_info` table stores the insurance coverage level for a customer. The BUSINESS_TIME period-related columns (`bus_start` and `bus_end`) indicate when an insurance coverage level is valid.

*Table 16. Created application-period temporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| | | | |

The `ix_policy` index, with BUSINESS_TIME WITHOUT OVERLAPS as the final column in the index key column list, ensures that there are no overlapping time periods for customer insurance coverage levels.

## Example

This section contains more examples of creating application-period temporal tables.

**Changing an existing table into an application-period temporal table**

The following example adds time columns and a BUSINESS_TIME period to an existing table (`employees`) enabling application-period temporal table functionality. Adding the BUSINESS_TIME WITHOUT OVERLAPS clause ensures that an employee is listed only once in any time period.

```
ALTER TABLE employees ADD COLUMN bus_start DATE NOT NULL;
ALTER TABLE employees ADD COLUMN bus_end DATE NOT NULL;
ALTER TABLE employees ADD PERIOD BUSINESS_TIME(bus_start, bus_end);
ALTER TABLE employees ADD CONSTRAINT uniq
   UNIQUE(employee_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

**Preventing overlapping periods of time**

In the “Procedure” section, an index ensures that there are no overlapping

BUSINESS_TIME periods. In the following alternative example, a PRIMARY KEY declaration is used when creating the `policy_info` table, ensuring that overlapping periods of BUSINESS_TIME are not allowed. This means that there cannot be two versions of the same policy that are valid at the same time.

```
CREATE TABLE policy_info
(
  policy_id     CHAR(4) NOT NULL,
  coverage      INT NOT NULL,
  bus_start     DATE NOT NULL,
  bus_end       DATE NOT NULL,
  PERIOD BUSINESS_TIME(bus_start, bus_end),
  PRIMARY KEY(policy_id, BUSINESS_TIME WITHOUT OVERLAPS)
);
```

**Ensuring uniqueness for periods of time**

The following example creates a `product_availability` table where a company tracks the products it distributes, the suppliers of those products, and the prices the suppliers charge. Multiple suppliers can provide the same product at the same time, but a PRIMARY KEY declaration ensures that a single supplier can only charge one price at any given point in time.

```
CREATE TABLE product_availability
(
  product_id     CHAR(4) NOT NULL,
  supplier_id    INT NOT NULL,
  product_price  DECIMAL NOT NULL
  bus_start      DATE NOT NULL,
  bus_end        DATE NOT NULL,
  PERIOD BUSINESS_TIME(bus_start, bus_end),
  PRIMARY KEY(product_id, supplier_id, BUSINESS_TIME WITHOUT OVERLAPS)
);
```

If the PRIMARY KEY was defined as

```
PRIMARY KEY(product_id, BUSINESS_TIME WITHOUT OVERLAPS)
```

then no two suppliers could deliver the same product at the same time.

# Dropping application-period temporal tables

Use the DROP TABLE statement to drop application-period temporal tables.

## About this task

When a table is dropped, the row in the SYSCAT.TABLES system catalog view that contains information about that table is dropped, and any other objects that depend on the table are affected. For example:

- All column names are dropped.
- Indexes created on any columns of the table are dropped.
- All views based on the table are marked inoperative.
- All privileges on the dropped table and dependent views are implicitly revoked.
- All referential constraints in which the table is a parent or dependent are dropped.
- All packages and cached dynamic SQL and XQuery statements dependent on the dropped table are marked invalid, and remain so until the dependent objects are re-created. This includes packages dependent on any supertable above the subtable in the hierarchy that is being dropped.
- Any reference columns for which the dropped table is defined as the scope of the reference become “unscoped”.

- An alias definition on the table is not affected, because an alias can be undefined
- All triggers dependent on the dropped table are marked inoperative.

Restrictions

An individual table cannot be dropped if it has a subtable.

### Procedure

- To drop a table, use a DROP TABLE statement.

  The following statement drops the table called DEPARTMENT:

  ```
  DROP TABLE DEPARTMENT
  ```

- To drop all the tables in a table hierarchy, use a DROP TABLE HIERARCHY statement.

  The DROP TABLE HIERARCHY statement must name the root table of the hierarchy to be dropped. For example:

  ```
  DROP TABLE HIERARCHY person
  ```

### Results

There are differences when dropping a table hierarchy compared to dropping a specific table:

- DROP TABLE HIERARCHY does not activate deletion-triggers that would be activated by individual DROP TABLE statements. For example, dropping an individual subtable would activate deletion-triggers on its supertables.
- DROP TABLE HIERARCHY does not make log entries for the individual rows of the dropped tables. Instead, the dropping of the hierarchy is logged as a single event.

## Bitemporal tables

A bitemporal table is a table that combines the historical tracking of a system-period temporal table with the time-specific data storage capabilities of an application-period temporal table. Use bitemporal tables to keep user-based period information as well as system-based historical information.

Bitemporal tables behave as a combination of system-period temporal tables and application-period temporal tables. All the restrictions that apply to system-period temporal tables and application temporal tables also apply to bitemporal tables.

## Creating a bitemporal table

Creating a bitemporal table results in a table that combines the historical tracking of a system-period temporal table with the time-specific data storage capabilities of an application-period temporal table.

### About this task

When creating a bitemporal table, you combine the steps used to create a system-period temporal table with the steps used to create an application-period temporal table.

- Include both a SYSTEM_TIME period and a BUSINESS_TIME period in the CREATE TABLE statement.
- Create a history table to receive old rows from the bitemporal table.

- Add versioning to establish the link between the bitemporal table and the history table.
- Optionally, define that overlapping periods of BUSINESS_TIME are not allowed and that values are unique with respect to any period.

The examples in the following section show the creation of a table that stores policy information for the customers of an insurance company.

## Procedure

To create a bitemporal table:

1. Create a table with both a SYSTEM_TIME attribute and a BUSINESS_TIME attribute. For example:

```
CREATE TABLE policy_info
(
 policy_id    CHAR(4) NOT NULL,
 coverage     INT NOT NULL,
 bus_start    DATE NOT NULL,
 bus_end      DATE NOT NULL,
 sys_start    TIMESTAMP(12) NOT NULL
              GENERATED ALWAYS AS ROW BEGIN,
 sys_end      TIMESTAMP(12) NOT NULL
              GENERATED ALWAYS AS ROW END,
 ts_id        TIMESTAMP(12) NOT NULL
              GENERATED ALWAYS AS TRANSACTION START ID,
 PERIOD BUSINESS_TIME (bus_start, bus_end),
 PERIOD SYSTEM_TIME (sys_start, sys_end)
) in policy_space;
```

2. Create a history table. For example:

```
CREATE TABLE hist_policy_info
(
 policy_id    CHAR(4) NOT NULL,
 coverage     INT NOT NULL,
 bus_start    DATE NOT NULL,
 bus_end      DATE NOT NULL,
 sys_start    TIMESTAMP(12) NOT NULL,
 sys_end      TIMESTAMP(12) NOT NULL,
 ts_id        TIMESTAMP(12)
) in hist_space;
```

   You can also create a history table with the same names and descriptions as the columns of the system-period temporal table using the LIKE clause of the CREATE TABLE statement. For example:

```
CREATE TABLE hist_policy_info LIKE policy_info in hist_space;
```

3. Add versioning to the bitemporal table. For example:

```
ALTER TABLE policy_info ADD VERSIONING USE HISTORY TABLE hist_policy_info;
```

4. Optional: Create a unique index that includes the BUSINESS_TIME period. For example:

```
CREATE UNIQUE INDEX ix_policy
   ON policy_info (policy_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

## Results

The `policy_info` table stores the insurance coverage level for a customer. The BUSINESS_TIME period-related columns (`bus_start` and `bus_end`) indicate when an insurance coverage level is valid. The SYSTEM_TIME period-related columns (`sys_start` and `sys_end`) show when a coverage level row is current. The `ts_id` column lists the time when execution started for a transaction that impacted the row.

*Table 17. Created bitemporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| | | | | | | |

The `hist_policy_info` history table receives the old rows from the `policy_info` table.

*Table 18. Created history table (hist_policy_info)*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| | | | | | | |

The `ix_policy` index, with BUSINESS_TIME WITHOUT OVERLAPS as the final column in the index key column list, ensures that there are no overlapping time periods for customer insurance coverage levels.

## Example

This section contains more creating bitemporal table examples.

**Hiding columns**

The following example creates the `policy_info` table with the TIMESTAMP(12) columns (`sys_start`, `sys_end` and `ts_id`) marked as implicitly hidden.

```
CREATE TABLE policy_info
(
 policy_id    CHAR(4) NOT NULL,
 coverage     INT NOT NULL,
 bus_start    DATE NOT NULL,
 bus_end      DATE NOT NULL,
 sys_start    TIMESTAMP(12) NOT NULL
              GENERATED ALWAYS AS ROW BEGIN IMPLICITLY HIDDEN,
 sys_end      TIMESTAMP(12) NOT NULL
              GENERATED ALWAYS AS ROW END IMPLICITLY HIDDEN,
 ts_id        TIMESTAMP(12)
              GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN,
 PERIOD BUSINESS_TIME (bus_start, bus_end),
 PERIOD SYSTEM_TIME   (sys_start, sys_end)
) in policy_space;
```

Creating the `hist_policy_info` history table using the LIKE clause of the CREATE TABLE statement results in the history table inheriting the implicitly hidden attribute from the `policy_info` table.

# Chapter 21. User-defined types

There are six types of user-defined data type.

- Distinct type
- Structured type
- Reference type
- Array type
- Row type
- Cursor type

Each of these types is described in the following sections.

## Distinct type

A *distinct type* is a user-defined data type that shares its internal representation with an existing type (its "source" type), but is considered to be a separate and incompatible type for most operations. For example, one might want to define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

The following example illustrates the creation of a distinct type named AUDIO:

```
CREATE TYPE AUDIO  AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type; this allows the creation of functions written specifically for AUDIO, and assures that these functions will not be applied to values of any other data type (pictures, text, and so on).

Distinct types have qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE TYPE (Distinct), DROP, or COMMENT statements, the SQL path is searched in sequence for the first schema with a distinct type that matches.

Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, because these may not be meaningful. (For example, the LENGTH function of the AUDIO type might return the length of its object in seconds rather than in bytes.)

Distinct types sourced on LOB types are subject to the same restrictions as their source type.

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type. This can be done by creating user-defined functions that are sourced on functions defined on the source type of the distinct type. The comparison operators are automatically generated for user-defined distinct types, except those using BLOB, CLOB, or DBCLOB as the source type. In addition, functions are generated to support casting from the source type to the distinct type, and from the distinct type to the source type.

## Structured type

A *structured type* is a user-defined data type that has a structure that is defined in the database. It contains a sequence of named *attributes*, each of which has a data type. A structured type also includes a set of method specifications.

A structured type may be used as the type of a table, view, or column. When used as a type for a table or view, that table or view is known as a *typed table* or *typed view*, respectively. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type. When used as a data type for a column, the column contains values of that structured type (or values of any of that type's subtypes, as defined later in this section). Methods are used to retrieve or manipulate attributes of a structured column object.

Terminology: A *supertype* is a structured type for which other structured types, called *subtypes*, have been defined. A subtype inherits all the attributes and methods of its supertype and may have additional attributes and methods defined. The set of structured types that are related to a common supertype is called a type *hierarchy* and the type that does not have any supertype is called the *root type* of the type hierarchy.

The term subtype applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. Therefore, a subtype of a structured type T is T and all structured types below T in the hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy.

There are restrictions on having recursive type definitions in a type hierarchy. For this reason, it is necessary to develop a shorthand way of referring to the specific type of recursive definitions that are allowed. The following definitions are used:

- *Directly uses*: A type **A** is said to directly use another type **B**, if and only if one of the following is true:
  1. type **A** has an attribute of type **B**
  2. type **B** is a subtype of **A**, or a supertype of **A**
- *Indirectly uses*: A type **A** is said to indirectly use a type **B**, if one of the following is true:
  1. type **A** directly uses type **B**
  2. type **A** directly uses some type **C**, and type **C** indirectly uses type **B**

A type may not be defined so that one of its attribute types directly or indirectly uses itself. If it is necessary to have such a configuration, consider using a reference as the attribute. For example, with structured type attributes, there cannot be an instance of "employee" with an attribute of "manager" when "manager" is of type "employee". There can, however, be an attribute of "manager" with a type of REF(employee).

A type cannot be dropped if certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes direct or indirect use of the type.

### Reference type

A *reference type* is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the *target type* of the reference.

The target of a reference is always a row in a typed table or a typed view. When a reference type is used, it may have a *scope* defined. The scope identifies a table (called the *target table*) or view (called the *target view*) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the *target row*.

### Array type

A user-defined *array type* is a data type that is defined as an array with elements of another data type. Every ordinary array type has an index with the data type of INTEGER and has a defined maximum cardinality. Every associative array has an index with the data type of INTEGER or VARCHAR and does not have a defined maximum cardinality.

### Row type

A *row type* is a data type that is defined as an ordered sequence of named fields, each with an associated data type, which effectively represents a row. A row type can be used as the data type for variables and parameters in SQL PL to provide simple manipulation of a row of data.

### Cursor type

A user-defined *cursor type* is a user-defined data type defined with the keyword CURSOR and optionally with an associated row type. A user-defined cursor type with an associated row type is a *strongly-typed cursor type*; otherwise, it is a *weakly-typed cursor type*. A value of a user-defined cursor type represents a reference to an underlying cursor.

## Distinct types

Distinct types are user-defined data types that are based on existing DB2 built-in data types. Internally, a distinct type shares its representation with an existing data type (the source data type), but is considered to be a separate and incompatible data type.

For example, distinct types can represent various currencies, such as `US_Dollar` or `Canadian_Dollar`. Both of these types are represented internally (and in your host language program) as the built-in data type upon which you defined these currencies. For example, if you define both currencies as DECIMAL, they are represented as decimal data types in the system.

DB2 also has built-in data types for storing and manipulating large objects. Your distinct type could be based on one of these large object (LOB) data types, which you might want to use for something like an audio or video stream. The following example illustrates the creation of a distinct type named AUDIO:

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate data type that is not comparable to a BLOB or to any other data type. This allows the creation of functions written specifically for AUDIO and assures that these functions will not be applied to any other data type.

**Restriction:** Not all built-in data types can be used to define distinct types. For example, neither the Boolean nor XML data types can be used to define a distinct type, nor can the array, row or cursor types. For more information, refer the documentation for the CREATE TYPE (Distinct) statement.

There are several benefits associated with distinct types:

1. Extensibility: By defining new data types, you can increase the set of types provided by DB2 to support your applications.
2. Flexibility: You can specify any semantics and behavior for your new data type by using user-defined functions (UDFs) to augment the diversity of the data types available in the system.
3. Consistent behavior: Strong typing insures that your distinct types will behave appropriately. It guarantees that only functions defined on your distinct type can be applied to instances of the distinct type.
4. Encapsulation: The set of functions and operators that you can apply to distinct types defines the behavior of your distinct types. This provides flexibility in the implementation since running applications do not depend on the internal representation that you choose for your data type.
5. Performance: Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code used to implement components such as built-in functions, comparison operators, and indexes for built-in data types.

Distinct types are identified by qualified identifiers. If the schema name is not used to qualify the distinct type name when used in statements other than CREATE DISTINCT TYPE, DROP DISTINCT TYPE, or COMMENT ON DISTINCT TYPE, the SQL path is searched in sequence for the first schema with a distinct type that matches.

Distinct types sourced on LONG VARCHAR, LONG VARGRAPHIC, or LOB data types are subject to the same restrictions as their source type. However, certain functions and operators of the source data type can be explicitly specified to apply to the distinct type by defining user-defined functions. (These functions are sourced on functions defined on the source data type of the distinct type.) The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, or DBCLOB as the source type. In addition, functions are generated to support casting from the source data type to the distinct type, and from the distinct type to the source data type.

## Creating distinct types

A user-defined distinct type is a data type derived from an existing type, such as an integer, decimal, or character type. To define a distinct type, you use the CREATE DISTINCT TYPE statement.

### Before you begin

Instances of the same distinct type can be compared to each other, if the WITH COMPARISONS clause is specified on the CREATE DISTINCT TYPE statement (as in the example in the procedure). If the source data type is a large object, LONG VARCHAR, or LONG VARGRAPHIC type, the WITH COMPARISONS clause cannot be specified.

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

### About this task

When you create distinct types, the database manager generates cast functions to cast from the distinct type to the source type, and to cast from the source type to the distinct type. These functions are essential for the manipulation of distinct types in queries. The source type of the distinct type is the data type used by the database manager to internally represent the distinct type. For this reason, it must be a built-in data type. Previously defined distinct types cannot be used as source types of other distinct types.

### Procedure

To define a distinct type, issue the CREATE DISTINCT TYPE statement, specifying a type name and the source type.
For example, the following statement defines a new distinct type called T_EDUC that contains SMALLINT values:

```
CREATE DISTINCT TYPE T_EDUC AS SMALLINT WITH COMPARISONS
```

Because the distinct type defined in the preceding statement is based on SMALLINT, the WITH COMPARISONS parameters must be specified.

### Results

After you create a distinct type, you can use it to define columns in a CREATE TABLE statement:

```
  CREATE TABLE EMPLOYEE
   (EMPNO      CHAR(6)      NOT NULL,
    FIRSTNME   VARCHAR(12)  NOT NULL,
    LASTNAME   VARCHAR(15)  NOT NULL,
    WORKDEPT   CHAR(3),
    PHONENO    CHAR(4),
    PHOTO      BLOB(10M)    NOT NULL,
    EDLEVEL    T_EDUC)
IN RESOURCE
```

## Creating tables with columns based on distinct types

After you define distinct types, you can start creating tables with columns based on distinct types.

### Before you begin

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

For the list of privileges required to create tables, see the CREATE TABLE statement.

### Procedure

To create a table with columns based on distinct types:

1. Define a distinct type:

```
CREATE DISTINCT TYPE t_educ AS SMALLINT WITH COMPARISONS
```

2. Create the table, naming the distinct type, T_EDUC as a column type.

```
CREATE TABLE employee
  (empno CHAR(6) NOT NULL,
   firstnme VARCHAR(12) NOT NULL,
   lastname VARCHAR(15) NOT NULL,
   workdept CHAR(3),
   phoneno CHAR(4),
   photo BLOB(10M) NOT NULL,
   edlevel T_EDUC)
  IN RESOURCE
```

## Creating currency-based distinct types

This topic describes how to create currency-based distinct types.

Suppose that you are writing applications that must handle different currencies. Given that conversions are necessary whenever you want to compare values of different currencies, you want to ensure that DB2 for Linux, UNIX, and Windows does not allow these currencies to be compared or manipulated directly with one another. Because distinct types are only compatible with themselves, you must define one for each currency that you need to represent.

### Before you begin

For the list of privileges required to define distinct types, see the CREATE DISTINCT TYPE statement.

### Procedure

To define distinct types representing the euro and the American and Canadian currencies, issue the following statements:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL (9,3) WITH COMPARISONS
```

Note that you must specify the WITH COMPARISONS clause because comparison operators are supported on DECIMAL (9,3).

## Casting between distinct types

This topic describes casting between distinct types.

### About this task

Suppose you want to define a UDF that converts another currency into U.S. dollars. For the purposes of this example, you can obtain the current exchange rate from a table such as the following query:

```
CREATE TABLE
  exchange_rates(source CHAR(3), target CHAR(3), rate DECIMAL(9,3))
```

The following function can be used to directly access the values in the exchange_rates table:

```
CREATE FUNCTION exchange_rate(src VARCHAR(3), trg VARCHAR(3))
  RETURNS DECIMAL(9,3)
  RETURN SELECT rate FROM exchange_rates
    WHERE source = src AND target = trg
```

The currency exchange rates in the preceding function are based on the DECIMAL type, not distinct types. To represent some different currencies, use the following distinct type definitions:

```
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL(9,3) WITH COMPARISONS
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
```

To create a UDF that converts CANADIAN_DOLLAR or EURO to US_DOLLAR you must cast the values involved. Note that the exchange_rate function returns an exchange rate as a DECIMAL. For example, a function that converts values of CANADIAN_DOLLAR to US_DOLLAR performs the following steps:

### Procedure

- cast the CANADIAN_DOLLAR value to DECIMAL
- get the exchange rate for converting the Canadian dollar to the U.S. dollar from the exchange_rate function, which returns the exchange rate as a DECIMAL value
- multiply the Canadian dollar DECIMAL value to the DECIMAL exchange rate
- cast this DECIMAL value to US_DOLLAR
- return the US_DOLLAR value

### What to do next

The following are instances of the US_DOLLAR function (for both the Canadian dollar and the euro), which follow the preceding steps.

```
CREATE FUNCTION US_DOLLAR(amount CANADIAN_DOLLAR)
  RETURNS US_DOLLAR
  RETURN US_DOLLAR(DECIMAL(amount) * exchange_rate('CAN', 'USD'))

CREATE FUNCTION US_DOLLAR(amount EURO)
  RETURNS US_DOLLAR
  RETURN US_DOLLAR(DECIMAL(amount) * exchange_rate('EUR', 'USD'))
```

## Dropping user-defined types

If you no longer need a user-defined type, you can drop it.

### About this task

You can drop a user-defined type (UDT) using the DROP statement. You cannot drop a UDT if it is used:

- In a column definition for an existing table or view.
- As the type of an existing typed table or typed view.
- As the supertype of another structured type.

The database manager attempts to drop every routine that is dependent on this UDT. A routine cannot be dropped if a view, trigger, table check constraint, or

another routine is dependent on it. If DB2 cannot drop a dependent routine, DB2 does not drop the UDT. Dropping a UDT invalidates any packages or cached dynamic SQL statements that used it.

If you have created a transform for a UDT, and you plan to drop that UDT, consider dropping the associated transform. To drop a transform, issue a DROP TRANSFORM statement. Note that you can only drop user-defined transforms. You cannot drop built-in transforms or their associated group definitions.

## Structured types

A structured type is a user-defined data type containing one or more named attributes, each of which has a data type. Attributes are properties that describe an instance of a type.

A structured type is a user-defined data type containing one or more named attributes, each of which has a data type. Attributes are properties that describe an instance of a type. A geometric shape, for example, might have attributes such as its list of Cartesian coordinates. A person might have attributes of name, address, and so on. A department might have attributes of a name or some other kind of ID.

A structured type also includes a set of method specifications. Methods enable you to define behaviors for structured types. Like user-defined functions (UDFs), methods are routines that extend SQL. In the case of methods, however, the behavior is integrated solely with a particular structured type.

A structured type can be used as the type of a table, view, or column. When used as the type for a table, that table is known as a typed table and when used as the type for a view, that view is known as a typed view. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of the typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type.

A type cannot be dropped when certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes a direct or indirect use of the type.

### Structured type hierarchies

This section provides information about the structured type hierarchies.

It is certainly possible to model objects such as people using traditional relational tables and columns. However, structured types offer an additional property of *inheritance*. That is, a structured type can have *subtypes* that reuse all of its attributes and contain additional attributes specific to the subtype. The original type is the *supertype*. For example, the structured type `Person_t` might contain attributes for `Name`, `Age`, and `Address`. A subtype of `Person_t` might be `Employee_t` that contains all of the attributes `Name`, `Age`, and `Address` and, in addition, contains attributes for `SerialNum`, `Salary`, and `BusinessUnit`.

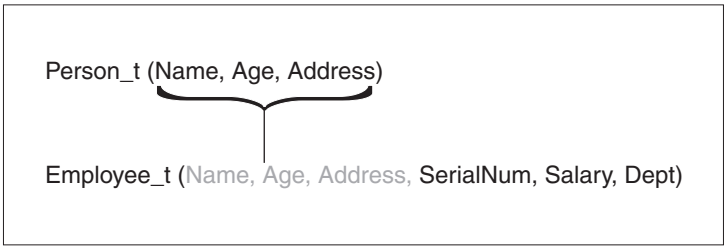



*Figure 11. Structured type Employee_t inherits attributes from supertype Person_t*

A set of subtypes based (at some level) on the same supertype is known as a type hierarchy. For example, a data model might need to represent a special type of employee called a manager. Managers have more attributes than employees who are not managers. The `Manager_t` type inherits the attributes defined for an employee, but also is defined with some additional attributes of its own, such as a special bonus attribute that is only available to managers.

The following figure presents an illustration of the various subtypes that might be derived from person and employee types:

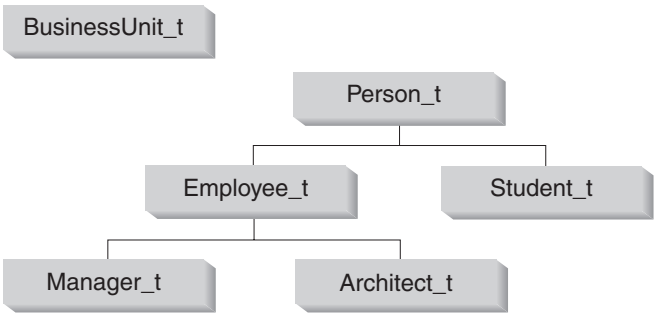



*Figure 12. Type hierarchies (BusinessUnit_t and Person_t)*

In Figure 12, the person type `Person_t` is the *root type* of the hierarchy. `Person_t` is also the supertype of the types below it--in this case, the type named `Employee_t` and the type named `Student_t`. The relationships among subtypes and supertypes are transitive; this means that the relationship between subtype and supertype exists throughout the entire type hierarchy. So, `Person_t` is also a supertype of types `Manager_t` and `Architect_t`.

The department type, `BusinessUnit_t` is considered a trivial type hierarchy. It is the root of a hierarchy with no subtypes.

## Creating structured types

This topic describes how to create structured types.

A structured type is a user-defined type that contains one or more attributes, each of which has a name and a data type of its own. A structured type can serve as the type of a table or view in which each column of the table derives its name and data type from one of the attributes of the structured type. A structured type can also serve as a type of a column or a type for an argument to a routine.

### Before you begin

For the list of privileges required to define structured types, see the CREATE TYPE statement.

### About this task

To define a structured type to represent a person, with age and address attributes, issue the following statement:

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   INSTANTIABLE
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;
```

Unlike distinct types, the attributes of structured types can be composed of types other than the built-in DB2 data types. The preceding type declaration includes an attribute called `Address` whose source type is another structured type, `Address_t`.

## Creating a structured type hierarchy

This topic describes how to create a structured type hierarchy.

### About this task

The following figure presents an illustration of a structured type hierarchy:

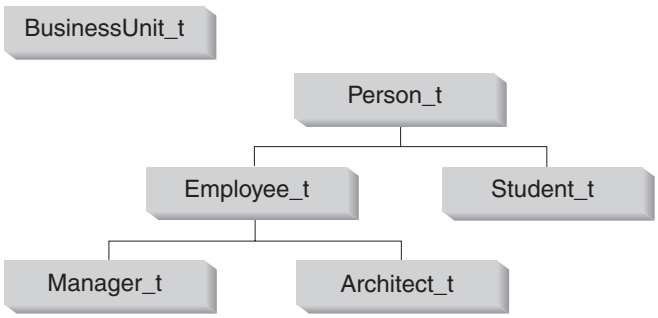



*Figure 13. Type hierarchies (BusinessUnit_t and Person_t)*

To create the `BusinessUnit_t` type, issue the following CREATE TYPE SQL statement:

```
CREATE TYPE BusinessUnit_t AS
   (Name VARCHAR(20),
   Headcount INT)
   MODE DB2SQL;
```

To create the `Person_t` type hierarchy, issue the following SQL statements:

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;

CREATE TYPE Employee_t UNDER Person_t AS
```

```
   (SerialNum INT,
   Salary DECIMAL(9,2),
   Dept REF(BusinessUnit_t))
   MODE DB2SQL;

CREATE TYPE Student_t UNDER Person_t AS
   (SerialNum CHAR(6),
   GPA DOUBLE)
   MODE DB2SQL;

CREATE TYPE Manager_t UNDER Employee_t AS
   (Bonus DECIMAL(7,2))
   MODE DB2SQL;

CREATE TYPE Architect_t UNDER Employee_t AS
   (StockOption INTEGER)
   MODE DB2SQL;
```

`Person_t` has three attributes: `Name`, `Age` and `Address`. Its two subtypes, `Employee_t` and `Student_t`, each inherit the attributes of `Person_t` and also have several additional attributes that are specific to their particular types. For example, although both employees and students have serial numbers, the format used for student serial numbers is different from the format used for employee serial numbers.

Finally, `Manager_t` and `Architect_t` are both subtypes of `Employee_t`; they inherit all the attributes of `Employee_t` and extend them further as appropriate for their types. Thus, an instance of type `Manager_t` will have a total of seven attributes: `Name`, `Age`, `Address`, `SerialNum`, `Salary`, `Dept`, and `Bonus`.

# Chapter 22. Constraints

Within any business, data must often adhere to certain restrictions or rules. For example, an employee number must be unique. The database manager provides *constraints* as a way to enforce such rules.

The following types of constraints are available:

- NOT NULL constraints
- Unique (or unique key) constraints
- Primary key constraints
- Foreign key (or referential integrity) constraints
- (Table) Check constraints
- Informational constraints

Constraints are only associated with tables and are either defined as part of the table creation process (using the CREATE TABLE statement) or are added to a table's definition after the table has been created (using the ALTER TABLE statement). You can use the ALTER TABLE statement to modify constraints. In most cases, existing constraints can be dropped at any time; this action does not affect the table's structure or the data stored in it.

**Note:** Unique and primary constraints are only associated with table objects, they are often enforced through the use of one or more unique or primary key indexes.

## Types of constraints

A *constraint* is a rule that is used for optimization purposes.

There are five types of constraints:

- A *NOT NULL constraint* is a rule that prevents null values from being entered into one or more columns within a table.
- A *unique constraint* (also referred to as a *unique key constraint*) is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint can be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not given to two suppliers.
- A *primary key constraint* is a column or combination of columns that has the same properties as a unique constraint. You can use a primary key and foreign key constraints to define relationships between tables.
- A *foreign key constraint* (also referred to as a *referential constraint* or a *referential integrity constraint*) is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's name changes. You can define a referential constraint stating that the ID of the supplier in a table must match a supplier ID in the supplier information. This constraint prevents insert, update, or delete operations that would otherwise result in missing supplier information.
- A *(table) check constraint* (also called a *check constraint*) sets restrictions on data added to a specific table. For example, a table check constraint can ensure that the salary level for an employee is at least $20 000 whenever salary data is added or updated in a table containing personnel information.

An *informational constraint* is an attribute of a certain type of constraint, but one that is not enforced by the database manager.

## NOT NULL constraints

NOT NULL constraints prevent null values from being entered into a column.

The null value is used in databases to represent an unknown state. By default, all of the built-in data types provided with the database manager support the presence of null values. However, some business rules might dictate that a value must always be provided (for example, every employee is required to provide emergency contact information). The NOT NULL constraint is used to ensure that a given column of a table is never assigned the null value. Once a NOT NULL constraint has been defined for a particular column, any insert or update operation that attempts to place a null value in that column will fail.

Because constraints only apply to a particular table, they are usually defined along with a table's attributes, during the table creation process. The following CREATE TABLE statement shows how the NOT NULL constraint would be defined for a particular column:

```
CREATE TABLE EMPLOYEES    (. . .
                           EMERGENCY_PHONE    CHAR(14)  NOT NULL,
                           . . .
                           );
```

## Unique constraints

*Unique constraints* ensure that the values in a set of columns are unique and not null for all rows in the table. The columns specified in a unique constraint must be defined as NOT NULL. The database manager uses a unique index to enforce the uniqueness of the key during changes to the columns of the unique constraint.

Unique constraints can be defined in the CREATE TABLE or ALTER TABLE statement using the UNIQUE clause. For example, a typical unique constraint in a DEPARTMENT table might be that the department number is unique and not null.

Figure 14 shows that a duplicate record is prevented from being added to a table when a unique constraint exists for the table:

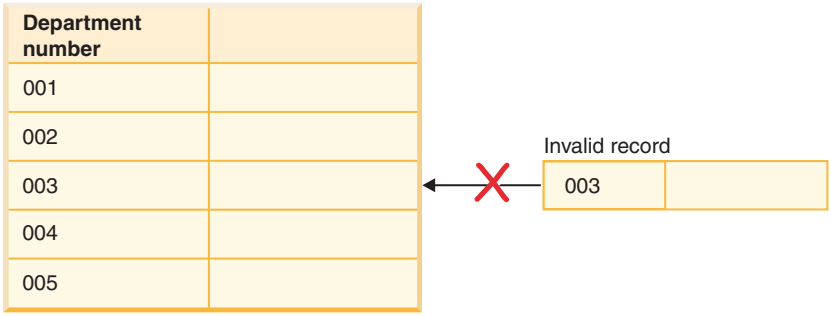



*Figure 14. Unique constraints prevent duplicate data*

The database manager enforces the constraint during insert and update operations, ensuring data integrity.

A table can have an arbitrary number of unique constraints, with at most one unique constraint defined as the primary key. A table cannot have more than one unique constraint on the same set of columns.

A unique constraint that is referenced by the foreign key of a referential constraint is called the *parent key*.

- When a unique constraint is defined in a CREATE TABLE statement, a unique index is automatically created by the database manager and designated as a primary or unique system-required index.
- When a unique constraint is defined in an ALTER TABLE statement and an index exists on the same columns, that index is designated as unique and system-required. If such an index does not exist, the unique index is automatically created by the database manager and designated as a primary or unique system-required index.

**Note:** There is a distinction between defining a unique constraint and creating a unique index. Although both enforce uniqueness, a unique index allows nullable columns and generally cannot be used as a parent key.

## Primary key constraints

You can use primary key and foreign key constraints to define relationships between tables.

A primary key is a column or combination of columns that has the same properties as a unique constraint. Because the primary key is used to identify a row in a table, it must be unique, and must have the NOT NULL attribute. A table cannot have more than one primary key, but it can have multiple unique keys. Primary keys are optional, and can be defined when a table is created or altered. They are also beneficial, because they order the data when data is exported or reorganized.

## (Table) Check constraints

A *check constraint* (also referred to as a *table check constraint*) is a database rule that specifies the values allowed in one or more columns of every row of a table. Specifying check constraints is done through a restricted form of a search condition.

### Designing check constraints

When creating check constraints, one of two things can happen: (i) all the rows meet the check constraint, or (ii) some or all the rows do not meet the check constraint.

#### About this task

**All the rows meet the check constraint**

When all the rows meet the check constraint, the check constraint will be created successfully. Future attempts to insert or update data that does not meet the constraint business rule will be rejected.

**Some or all the rows do not meet the check constraint**

When there are some rows that do not meet the check constraint, the check constraint will not be created (that is, the ALTER TABLE statement will fail). The ALTER TABLE statement, which adds a new constraint to the EMPLOYEE table, is shown in the following example. The check constraint is named CHECK_JOB. The database manager will use this name to inform

you about which constraint was violated if an INSERT or UPDATE statement fails. The CHECK clause is used to define a table-check constraint.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT check_job
  CHECK (JOB IN ('Engineer', 'Sales', 'Manager'));
```

An ALTER TABLE statement was used because the table had already been defined. If there are values in the EMPLOYEE table that conflict with the constraint being defined, the ALTER STATEMENT will not be completed successfully.

As check constraints and other types of constraints are used to implement business rules, you might need to change them from time to time. This could happen when the business rules change in your organization. Whenever a check constraint needs to be changed, you must drop it and re-create a new one. Check constraints can be dropped at any time, and this action will not affect your table or the data within it. When you drop a check constraint, you must be aware that data validation performed by the constraint will no longer be in effect.

## Comparison of check constraints and BEFORE triggers

You must consider the difference between check constraints when considering whether to use triggers or check constraints to preserve the integrity of your data.

The integrity of the data in a relational database must be maintained as multiple users access and change the data. Whenever data is shared, there is a need to ensure the accuracy of the values within databases.

**Check constraints**

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

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

The enforcement of check constraints is important for maintaining data integrity, but it also carries a certain amount of system activity that can impact performance whenever large volumes of data are modified.

**BEFORE triggers**

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

In addition to modification, a common use of the BEFORE triggers is for data verification using the SIGNAL clause.

There are two differences between BEFORE triggers and check constraints when used for data verification:

1. BEFORE triggers, unlike check constraints, are not restricted to access other values in the same row of the same table.
2. During a SET INTEGRITY operation on a table after a LOAD operation, triggers (including BEFORE triggers) are not executed. Check constraints, however, are verified.

## Foreign key (referential) constraints

*Foreign key constraints* (also known as *referential constraints* or *referential integrity constraints*) enable you to define required relationships between and within tables.

For example, a typical foreign key constraint might state that every employee in the EMPLOYEE table must be a member of an existing department, as defined in the DEPARTMENT table.

*Referential integrity* is the state of a database in which all values of all foreign keys are valid. A *foreign key* is a column or a set of columns in a table whose values are required to match at least one primary key or unique key value of a row in its parent table. A *referential constraint* is the rule that the values of the foreign key are valid only if one of the following conditions is true:

- They appear as values of a parent key.
- Some component of the foreign key is null.

To establish this relationship, you would define the department number in the EMPLOYEE table as the foreign key, and the department number in the DEPARTMENT table as the primary key.

Figure 15 on page 178 shows how a record with an invalid key is prevented from being added to a table when a foreign key constraint exists between two tables:

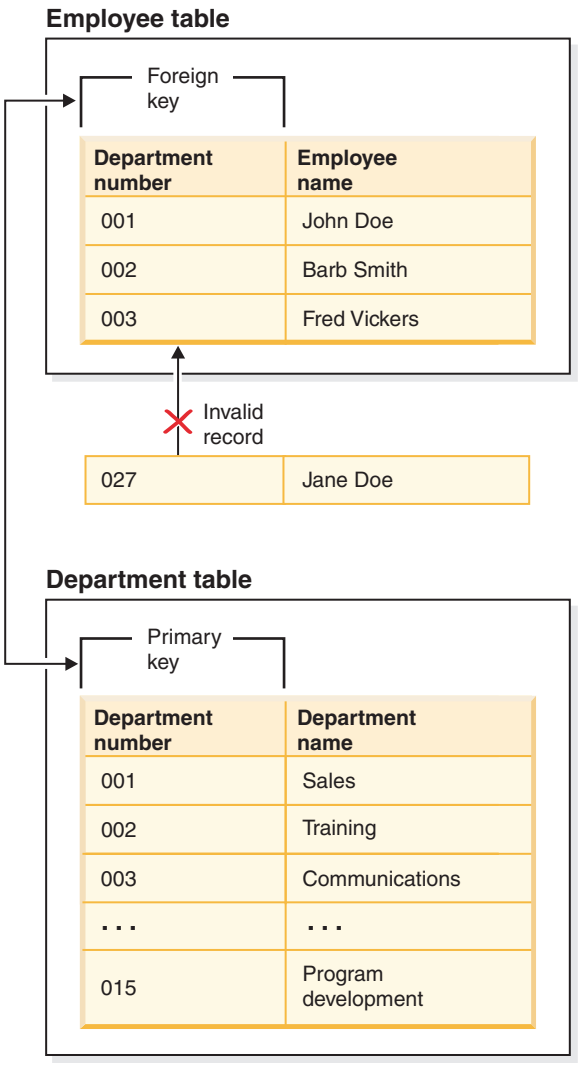



*Figure 15. Foreign and primary key constraints*

The table containing the parent key is called the *parent table* of the referential constraint, and the table containing the foreign key is said to be a *dependent* of that table.

Referential constraints can be defined in the CREATE TABLE statement or the ALTER TABLE statement. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE, MERGE, ADD CONSTRAINT, and SET INTEGRITY statements.

Referential integrity rules involve the following terms:

*Table 19. Referential integrity terms*

| Concept | Terms |
|---|---|
| Parent key | A primary key or a unique key of a referential constraint. |
| Parent row | A row that has at least one dependent row. |
| Parent table | A table that contains the parent key of a referential constraint. A table can be a parent in an arbitrary number of referential constraints. A table that is the parent in a referential constraint can also be the dependent in a referential constraint. |

*Table 19. Referential integrity terms (continued)*

| Concept | Terms |
|---|---|
| Dependent table | A table that contains at least one referential constraint in its definition. A table can be a dependent in an arbitrary number of referential constraints. A table that is the dependent in a referential constraint can also be the parent in a referential constraint. |
| Descendent table | A table is a descendent of table T if it is a dependent of T or a descendent of a dependent of T. |
| Dependent row | A row that has at least one parent row. |
| Descendent row | A row is a descendent of row r if it is a dependent of r or a descendent of a dependent of r. |
| Referential cycle | A set of referential constraints such that each table in the set is a descendent of itself. |
| Self-referencing table | A table that is a parent and a dependent in the same referential constraint. The constraint is called a *self-referencing constraint*. |
| Self-referencing row | A row that is a parent of itself. |

The purpose of a referential constraint is to guarantee that table relationships are maintained and that data entry rules are followed. This means that as long as a referential constraint is in effect, the database manager guarantees that for each row in a child table that has a non-null value in its foreign key columns, a row exists in a corresponding parent table that has a matching value in its parent key.

When an SQL operation attempts to change data in such a way that referential integrity will be compromised, a foreign key (or referential) constraint could be violated. The database manager handles these types of situations by enforcing a set of rules that are associated with each referential constraint. This set of rules consist of:

- An insert rule
- An update rule
- A delete rule

When an SQL operation attempts to change data in such a way that referential integrity will be compromised, a referential constraint could be violated. For example,

- An insert operation could attempt to add a row of data to a child table that has a value in its foreign key columns that does not match a value in the corresponding parent table's parent key.
- An update operation could attempt to change the value in a child table's foreign key columns to a value that has no matching value in the corresponding parent table's parent key.
- An update operation could attempt to change the value in a parent table's parent key to a value that does not have a matching value in a child table's foreign key columns.
- A delete operation could attempt to remove a record from a parent table that has a matching value in a child table's foreign key columns.

The database manager handles these types of situations by enforcing a set of rules that are associated with each referential constraint. This set of rules consists of:

- An insert rule

- An update rule
- A delete rule

## Insert rule

The insert rule of a referential constraint is that a non-null insert value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is null if any component of the value is null. This rule is implicit when a foreign key is specified.

## Update rule

The update rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION and RESTRICT. The update rule applies when a row of the parent or a row of the dependent table is updated.

In the case of a parent row, when a value in a column of the parent key is updated, the following rules apply:

- If any row in the dependent table matches the original value of the key, the update is rejected when the update rule is RESTRICT.
- If any row in the dependent table does not have a corresponding parent key when the update statement is completed (excluding AFTER triggers), the update is rejected when the update rule is NO ACTION.

The value of the parent unique keys cannot be changed if the update rule is RESTRICT and there are one or more dependent rows. However, if the update rule is NO ACTION, parent unique keys can be updated as long as every child has a parent key by the time the update statement completes. A non-null update value of a foreign key must be equal to a value of the primary key of the parent table of the relationship.

Also, the use of NO ACTION or RESTRICT as update rules for referential constraints determines when the constraint is enforced. An update rule of RESTRICT is enforced before all other constraints, including those referential constraints with modifying rules such as CASCADE or SET NULL. An update rule of NO ACTION is enforced after other referential constraints. Note that the SQLSTATE returned is different depending on whether the update rule is RESTRICT or NO ACTION.

In the case of a dependent row, the NO ACTION update rule is implicit when a foreign key is specified. NO ACTION means that a non-null update value of a foreign key must match some value of the parent key of the parent table when the update statement is completed.

The value of a composite foreign key is null if any component of the value is null.

## Delete rule

The delete rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

If the identified table or the base table of the identified view is a parent, the rows selected for delete must not have any dependents in a relationship with a delete

rule of RESTRICT, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT.

If the delete operation is not prevented by a RESTRICT delete rule, the selected rows are deleted. Any rows that are dependents of the selected rows are also affected:

- The nullable columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET NULL are set to the null value.
- Any rows that are their dependents in a relationship with a delete rule of CASCADE are also deleted, and the rules mentioned previously apply, in turn, to those rows.

The delete rule of NO ACTION is checked to enforce that any non-null foreign key refers to an existing parent row after the other referential constraints have been enforced.

The delete rule of a referential constraint applies only when *a row* of the parent table is deleted. More precisely, the rule applies only when *a row* of the parent table is the object of a delete or propagated delete operation (defined in the following section), and that row has dependents in the dependent table of the referential constraint. Consider an example where P is the parent table, D is the dependent table, and p is a parent row that is the object of a delete or propagated delete operation. The delete rule works as follows:

- With RESTRICT or NO ACTION, an error occurs and no rows are deleted.
- With CASCADE, the delete operation is propagated to the dependents of p in table D.
- With SET NULL, each nullable column of the foreign key of each dependent of p in table D is set to null.

Any table that can be involved in a delete operation on P is said to be *delete-connected* to P. Thus, a table is delete-connected to table P if it is a dependent of P, or a dependent of a table to which delete operations from P cascade.

The following restrictions apply to delete-connected relationships:

- When a table is delete-connected to itself in a referential cycle of more than one table, the cycle must not contain a delete rule of either RESTRICT or SET NULL.
- A table must not both be a dependent table in a CASCADE relationship (self-referencing or referencing another table) and have a self-referencing relationship with a delete rule of either RESTRICT or SET NULL.
- When a table is delete-connected to another table through multiple relationships where such relationships have overlapping foreign keys, these relationships must have the same delete rule and none of these can be SET NULL.
- When a table is delete-connected to another table through multiple relationships where one of the relationships is specified with delete rule SET NULL, the foreign key definition of this relationship must not contain any distribution key or MDC key column, a table-partitioning key column, or RCT key column.
- When two tables are delete-connected to the same table through CASCADE relationships, the two tables must not be delete-connected to each other where the delete connected paths end with delete rule RESTRICT or SET NULL.

## Examples of interaction between triggers and referential constraints

Update operations can cause the interaction of triggers with referential constraints and check constraints.

Figure 16 and the associated description are representative of the processing that is performed for an statement that updates data in the database.

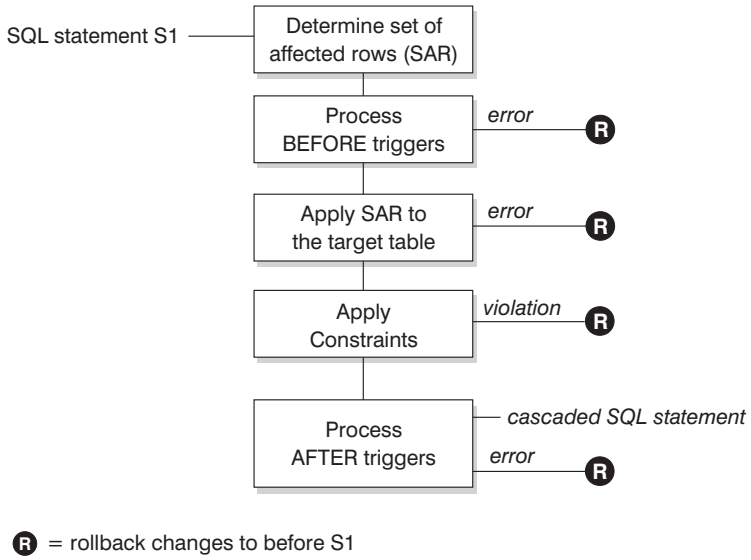



*Figure 16. Processing an statement with associated triggers and constraints*

Figure 16 shows the general order of processing for an statement that updates a table. It assumes a situation where the table includes BEFORE triggers, referential constraints, check constraints and AFTER triggers that cascade. The following is a description of the boxes and other items found in Figure 16.

- statement $S_1$

  This is the DELETE, INSERT, or UPDATE statement that begins the process. The statement $S_1$ identifies a table (or an updatable view over some table) referred to as the *subject table* throughout this description.

- Determine set of affected rows

  This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded statements from AFTER triggers.

  The purpose of this step is to determine the *set of affected rows* for the statement. The set of rows included is based on the statement:

  – for DELETE, all rows that satisfy the search condition of the statement (or the current row for a positioned DELETE)

  – for INSERT, the rows identified by the VALUES clause or the fullselect

  – for UPDATE, all rows that satisfy the search condition (or the current row for a positioned UPDATE).

  If the set of affected rows is empty, there will be no BEFORE triggers, changes to apply to the subject table, or constraints to process for the statement.

- Process BEFORE triggers

All BEFORE triggers are processed in ascending order of creation. Each BEFORE trigger will process the triggered action once for each row in the set of affected rows.

An error can occur during the processing of a triggered action in which case all changes made as a result of the original statement $S_1$ (so far) are rolled back.

If there are no BEFORE triggers or the set of affected is empty, this step is skipped.

- Apply the set of affected rows to the subject table

  The actual delete, insert, or update is applied using the set of affected rows to the subject table in the database.

  An error can occur when applying the set of affected rows (such as attempting to insert a row with a duplicate key where a unique index exists) in which case all changes made as a result of the original statement $S_1$ (so far) are rolled back.

- Apply Constraints

  The constraints associated with the subject table are applied if set of affected rows is not empty. This includes unique constraints, unique indexes, referential constraints, check constraints and checks related to the WITH CHECK OPTION on views. Referential constraints with delete rules of cascade or set null might cause additional triggers to be activated.

  A violation of any constraint or WITH CHECK OPTION results in an error and all changes made as a result of $S_1$ (so far) are rolled back.

- Process AFTER triggers

  All AFTER triggers activated by $S_1$ are processed in ascending order of creation.

  FOR EACH STATEMENT triggers will process the triggered action exactly once, even if the set of affected rows is empty. FOR EACH ROW triggers will process the triggered action once for each row in the set of affected rows.

  An error can occur during the processing of a triggered action in which case all changes made as a result of the original $S_1$ (so far) are rolled back.

  The triggered action of a trigger can include triggered statements that are DELETE, INSERT or UPDATE statements. For the purposes of this description, each such statement is considered a *cascaded statement*.

  A cascaded statement is a DELETE, INSERT, or UPDATE statement that is processed as part of the triggered action of an AFTER trigger. This statement starts a cascaded level of trigger processing. This can be thought of as assigning the triggered statement as a new $S_1$ and performing all of the steps described here recursively.

  Once all triggered statements from all AFTER triggers activated by each $S_1$ have been processed to completion, the processing of the original $S_1$ is completed.

- R = roll back changes to before $S_1$

  Any error (including constraint violations) that occurs during processing results in a roll back of all the changes made directly or indirectly as a result of the original statement $S_1$. The database is therefore back in the same state as immediately before the execution of the original statement $S_1$

## Informational constraints

An *informational constraint* is a constraint attribute that can be used by the SQL compiler to improve the access to data. Informational constraints are not enforced by the database manager, and are not used for additional verification of data; rather, they are used to improve query performance.

Informational constraints are defined using the CREATE TABLE or ALTER TABLE statements. You first add referential integrity or check constraints and then associate constraint attributes to them specifying whether the database manager is to enforce the constraint or not. For check constraints you can further specify that the constraint can be trusted. For referential integrity constraints, if the constraint is not enforced, you can further specify whether the constraint can be trusted or not. A not enforced and not trusted constraint is also known as a statistical referential integrity constraint. After you have specified the constraint you can then specify whether the constraint is to be used for query optimization or not.

Informational RI (referential integrity) constraints are used to optimize query performance, the incremental processing of REFRESH IMMEDIATE MQT, and staging tables. Query results, MQT data, and staging tables might be incorrect if informational constraints are violated.

For example, the order in which parent-child tables are maintained is important. When you want to add rows to a parent-child table, you must insert rows into the parent table first. To remove rows from a parent-child table, you must delete rows from the child table first. This ensures that there are no orphan rows in the child table at any time. Otherwise the informational constraint violation might affect the correctness of queries being executed during table maintenance, as well as the correctness of the incremental maintenance of dependent MQT data and staging tables.

## Designing informational constraints

Constraints that are enforced by the database manager when records are inserted or updated can lead to high amounts of system activity, especially when loading large quantities of records that have referential integrity constraints. If an application has already verified information before inserting a record into the table, it might be more efficient to use informational constraints, rather than normal constraints.

Informational constraints tell the database manager what rules the data conforms to, but the rules are not enforced by the database manager. However, this information can be used by the DB2 optimizer and could result in better performance of SQL queries.

The following example illustrates the use of information constraints and how they work. This simple table contains information about applicants' age and gender:

```
CREATE TABLE APPLICANTS
  (
   AP_NO INT NOT NULL,
     GENDER  CHAR(1) NOT NULL,
CONSTRAINT GENDEROK
   CHECK (GENDER IN ('M', 'F'))
   NOT ENFORCED
   ENABLE QUERY OPTIMIZATION,
      AGE  INT NOT NULL,
CONSTRAINT AGEOK
   CHECK (AGE BETWEEN 1 AND 80)
   NOT ENFORCED
   ENABLE QUERY OPTIMIZATION,
 );
```

This example contains two options that change the behavior of the column constraints. The first option is NOT ENFORCED, which instructs the database manager not to enforce the checking of this column when data is inserted or

updated. This option can be further specified to be either TRUSTED or NOT TRUSTED. If the informational constraint is specified to be TRUSTED then the database manager can trust that the data will conform to the constraint. This is the default option. If NOT TRUSTED is specified then the database manager knows that most of the data, but not all, will not conform to the constraint. In this example, the option is NOT ENFORCED TRUSTED by default since the option of trusted or not trusted was not specified.

The second option is ENABLE QUERY OPTIMIZATION which is used by the database manager when SELECT statements are run against this table. When this value is specified, the database manager will use the information in the constraint when optimizing the SQL.

If the table contains the NOT ENFORCED option, the behavior of insert statements might appear odd. The following SQL will not result in any errors when run against the APPLICANTS table:

```
INSERT INTO APPLICANTS VALUES
 (1, 'M', 54),
 (2, 'F', 38),
 (3, 'M', 21),
 (4, 'F', 89),
 (5, 'C', 10),
 (6, 'S',100),
```

Applicant number five has a gender (C), for child, and applicant number six has both an unusual gender and exceeds the age limits of the AGE column. In both cases the database manager will allow the insert to occur since the constraints are NOT ENFORCED and TRUSTED. The result of a select statement against the table is shown in the following example:

```
SELECT * FROM APPLICANTS
  WHERE GENDER = 'C';

APPLICANT  GENDER  AGE
---------  ------  ---

0 record(s) selected.
```

The database manager returned the incorrect answer to the query, even though the value 'C' is found within the table, but the constraint on this column tells the database manager that the only valid values are either 'M' or 'F'. The ENABLE QUERY OPTIMIZATION keyword also allowed the database manager to use this constraint information when optimizing the statement. If this is not the behavior that you want, then the constraint needs to be changed through the use of the ALTER TABLE statement, as shown in the following example:

```
ALTER TABLE APPLICANTS
  ALTER CHECK AGEOK DISABLE QUERY OPTIMIZATION
```

If the query is reissued, the database manager will return the following correct results:

```
SELECT * FROM APPLICANTS
  WHERE SEC = 'C';

APPLICANT GENDER  AGE
---------  ------  ---
        5  C       10

1 record(s) selected.
```

**Note:** If the constraint attributes NOT ENFORCED NOT TRUSTED and ENABLE QUERY OPTIMIZATION were specified from the beginning for the table APPLICANTS, then the correct results shown previously would have been returned after the first SELECT statement was issued.

The best scenario for using NOT ENFORCED TRUSTED informational constraints occurs when you can guarantee that the application program is the only application inserting and updating the data. If the application already checks all of the information beforehand (such as gender and age in the previous example) then using informational constraints can result in faster performance and no duplication of effort. Another possible use of informational constraints is in the design of data warehouses. Also, if you cannot guarantee that the data in the table will always conform to the constraint you can set the constraints to be NOT ENFORCED and NOT TRUSTED. This type of constraint can be used when strict matching between the values in the foreign keys and the primary keys are not needed. This constraint can also still be used as part of a statistical view enabling the optimization of certain SQL queries.

## Creating and modifying constraints

Constraints can be added to existing tables with the ALTER TABLE statement.

### About this task

The constraint name cannot be the same as any other constraint specified within an ALTER TABLE statement, and must be unique within the table (this includes the names of any referential integrity constraints that are defined). Existing data is checked against the new condition before the statement succeeds.

**Creating and modifying unique constraints**

Unique constraints can be added to an existing table. The constraint name cannot be the same as any other constraint specified within the ALTER TABLE statement, and must be unique within the table (this includes the names of any referential integrity constraints that are defined). Existing data is checked against the new condition before the statement succeeds.

To define unique constraints using the command line, use the `ADD CONSTRAINT` option of the ALTER TABLE statement. For example, the following statement adds a unique constraint to the EMPLOYEE table that represents a new way to uniquely identify employees in the table:

```
ALTER TABLE EMPLOYEE
   ADD CONSTRAINT NEWID UNIQUE(EMPNO,HIREDATE)
```

To modify this constraint, you would have to drop it, and then re-create it.

**Creating and modifying primary key constraints**

A primary key constraint can be added to an existing table. The constraint name must be unique within the table (this includes the names of any referential integrity constraints that are defined). Existing data is checked against the new condition before the statement succeeds.

To add primary keys using the command line, enter:

```
ALTER TABLE <name>
   ADD CONSTRAINT <column_name>
   PRIMARY KEY <column_name>
```

An existing constraint cannot be modified. To define another column, or set of columns, as the primary key, the existing primary key definition must first be dropped, and then re-created.

**Creating and modifying check constraints**

When a table check constraint is added, packages and cached dynamic SQL that insert or update the table might be marked as invalid.

To add a table check constraint using the command line, enter:

```
ALTER TABLE EMPLOYEE
   ADD CONSTRAINT REVENUE CHECK (SALARY + COMM > 25000)
```

To modify this constraint, you would have to drop it, and then re-create it.

**Creating and modifying foreign key (referential) constraints**

A foreign key is a reference to the data values in another table. There are different types of foreign key constraints.

When a foreign key is added to a table, packages and cached dynamic SQL containing the following statements might be marked as invalid:

- Statements that insert or update the table containing the foreign key
- Statements that update or delete the parent table.

To add foreign keys using the command line, enter:

```
ALTER TABLE <name>
   ADD CONSTRAINT <column_name>
    FOREIGN KEY <column_name>
    ON DELETE <action_type>
    ON UPDATE <action_type>
```

The following examples show the ALTER TABLE statement to add primary keys and foreign keys to a table:

```
ALTER TABLE PROJECT
  ADD CONSTRAINT PROJECT_KEY
      PRIMARY KEY (PROJNO)
ALTER TABLE EMP_ACT
  ADD CONSTRAINT ACTIVITY_KEY
      PRIMARY KEY (EMPNO, PROJNO, ACTNO)
  ADD CONSTRAINT ACT_EMP_REF
      FOREIGN KEY (EMPNO)
      REFERENCES EMPLOYEE
      ON DELETE  RESTRICT
  ADD CONSTRAINT ACT_PROJ_REF
      FOREIGN KEY (PROJNO)
      REFERENCES PROJECT
      ON DELETE CASCADE
```

To modify this constraint, you would have to drop it and then re-create it.

**Creating and modifying informational constraints**

To improve the performance of queries, you can add informational constraints to your tables. You add informational constraints using the CREATE TABLE or ALTER TABLE statement when you specify the `NOT ENFORCED` option on the DDL. Along with the NOT ENFORCED option you can further specify the constraint to be either TRUSTED or NOT TRUSTED.

**Restriction:** After you define informational constraints on a table, you can only alter the column names for that table after you remove the informational constraints.

To specify informational constraints on a table using the command line, enter one of the following commands for a new table:

```
ALTER TABLE <name> <constraint attributes> NOT ENFORCED
ALTER TABLE <name> <constraint attributes> NOT ENFORCED TRUSTED
ALTER TABLE <name> <constraint attributes> NOT ENFORCED NOT TRUSTED
```

ENFORCED or NOT ENFORCED: Specifies whether the constraint is enforced by the database manager during normal operations such as insert, update, or delete.

- ENFORCED cannot be specified for a functional dependency (SQLSTATE 42621).
- NOT ENFORCED should only be specified if the table data is independently known to conform to the constraint. Query results might be unpredictable if the data does not actually conform to the constraint. You can also specify if the NOT ENFORCED constraint is to be TRUSTED or NOT TRUSTED.
  - TRUSTED: Informs the database manager that the data can be trusted to conform to the constraint. This is the default option. This option must only be used if the data is independently known to conform to the constraint
  - NOT TRUSTED: Informs the database manager that the data cannot be trusted to conform to the constraint. This option is intended for cases where the data conforms to the constraint for most rows, but it is not independently known to conform to the constraint. NOT TRUSTED can be specified only for referential integrity constraints (SQLSTATE 42613).

To modify this constraint, you would have to drop it and then re-create it.

## Table constraint implications for utility operations

If the table being loaded into has referential integrity constraints, the load utility places the table into the set integrity pending state to inform you that the SET INTEGRITY statement is required to be run on the table, in order to verify the referential integrity of the loaded rows. After the load utility has completed, you will need to issue the SET INTEGRITY statement to carry out the referential integrity checking on the loaded rows and to bring the table out of the set integrity pending state.

For example, if the DEPARTMENT and EMPLOYEE tables are the only tables that have been placed in set integrity pending state, you can execute the following statement:

```
SET INTEGRITY FOR DEPARTMENT ALLOW WRITE ACCESS,
   EMPLOYEE ALLOW WRITE ACCESS,
   IMMEDIATE CHECKED FOR EXCEPTION IN DEPARTMENT,
   USE DEPARTMENT_EX,
   IN EMPLOYEE USE EMPLOYEE_EX
```

The import utility is affected by referential constraints in the following ways:

- The REPLACE and REPLACE CREATE functions are not allowed if the object table has any dependents other than itself.

  To use these functions, first drop all foreign keys in which the table is a parent. When the import is complete, re-create the foreign keys with the ALTER TABLE statement.

- The success of importing into a table with self-referencing constraints depends on the order in which the rows are imported.

## Checking for integrity violations following a load operation

Following a load operation, the loaded table might be in set integrity pending state in either READ or NO ACCESS mode if any of the following conditions exist:

- The table has table check constraints or referential integrity constraints defined on it.
- The table has generated columns and a V7 or earlier client was used to initiate the load operation.
- The table has descendent immediate materialized query tables or descendent immediate staging tables referencing it.
- The table is a staging table or a materialized query table.

The `STATUS` flag of the `SYSCAT.TABLES` entry corresponding to the loaded table indicates the set integrity pending state of the table. For the loaded table to be fully usable, the `STATUS` must have a value of `N` and the `ACCESS MODE` must have a value of `F`, indicating that the table is fully accessible and in normal state.

If the loaded table has descendent tables, the SET INTEGRITY PENDING CASCADE parameter can be specified to indicate whether or not the set integrity pending state of the loaded table should be immediately cascaded to the descendent tables.

If the loaded table has constraints as well as descendent foreign key tables, dependent materialized query tables and dependent staging tables, and if all of the tables are in normal state before the load operation, the following will result based on the load parameters specified:

**INSERT, ALLOW READ ACCESS, and SET INTEGRITY PENDING CASCADE IMMEDIATE**

The loaded table, its dependent materialized query tables and dependent staging tables are placed in set integrity pending state with read access.

**INSERT, ALLOW READ ACCESS, and SET INTEGRITY PENDING CASCADE DEFERRED**

Only the loaded table is placed in set integrity pending with read access. Descendent foreign key tables, descendent materialized query tables and descendent staging tables remain in their original states.

**INSERT, ALLOW NO ACCESS, and SET INTEGRITY PENDING CASCADE IMMEDIATE**

The loaded table, its dependent materialized query tables and dependent staging tables are placed in set integrity pending state with no access.

**INSERT or REPLACE, ALLOW NO ACCESS, and SET INTEGRITY PENDING CASCADE DEFERRED**

Only the loaded table is placed in set integrity pending state with no access. Descendent foreign key tables, descendent immediate materialized query tables and descendent immediate staging tables remain in their original states.

**REPLACE, ALLOW NO ACCESS, and SET INTEGRITY PENDING CASCADE IMMEDIATE**

The table and all its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables are placed in set integrity pending state with no access.

**Note:** Specifying the ALLOW READ ACCESS option in a load replace operation results in an error.

To remove the set integrity pending state, use the SET INTEGRITY statement. The SET INTEGRITY statement checks a table for constraints violations, and takes the table out of set integrity pending state. If all the load operations are performed in INSERT mode, the SET INTEGRITY statement can be used to incrementally process the constraints (that is, it checks only the appended portion of the table for constraints violations). For example:

```
db2 load from infile1.ixf of ixf insert into table1
db2 set integrity for table1 immediate checked
```

Only the appended portion of TABLE1 is checked for constraint violations. Checking only the appended portion for constraints violations is faster than checking the entire table, especially in the case of a large table with small amounts of appended data.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for setting integrity. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

If a table is loaded with the SET INTEGRITY PENDING CASCADE DEFERRED option specified, and the SET INTEGRITY statement is used to check for integrity violations, the descendent tables are placed in set integrity pending state with no access. To take the tables out of this state, you must issue an explicit request.

If a table with dependent materialized query tables or dependent staging tables is loaded using the INSERT option, and the SET INTEGRITY statement is used to check for integrity violations, the table is taken out of set integrity pending state and placed in No Data Movement state. This is done to facilitate the subsequent incremental refreshes of the dependent materialized query tables and the incremental propagation of the dependent staging tables. In the No Data Movement state, operations that might cause the movement of rows within the table are not allowed.

You can override the No Data Movement state by specifying the FULL ACCESS option when you issue the SET INTEGRITY statement. The table is fully accessible, however a full re-computation of the dependent materialized query tables takes place in subsequent REFRESH TABLE statements and the dependent staging tables are forced into an incomplete state.

If the ALLOW READ ACCESS option is specified for a load operation, the table remains in read access state until the SET INTEGRITY statement is used to check for constraints violations. Applications can query the table for data that existed before the load operation once it has been committed, but will not be able to view the newly loaded data until the SET INTEGRITY statement is issued.

Several load operations can take place on a table before checking for constraints violations. If all of the load operations are completed in ALLOW READ ACCESS mode, only the data that existed in the table before the first load operation is available for queries.

One or more tables can be checked in a single invocation of this statement. If a dependent table is to be checked on its own, the parent table can not be in set

integrity pending state. Otherwise, both the parent table and the dependent table must be checked at the same time. In the case of a referential integrity cycle, all the tables involved in the cycle must be included in a single invocation of the SET INTEGRITY statement. It might be convenient to check the parent table for constraints violations while a dependent table is being loaded. This can only occur if the two tables are not in the same table space.

When issuing the SET INTEGRITY statement, you can specify the INCREMENTAL option to explicitly request incremental processing. In most cases, this option is not needed, because the DB2 database selects incremental processing. If incremental processing is not possible, full processing is used automatically. When the INCREMENTAL option is specified, but incremental processing is not possible, an error is returned if:

- New constraints are added to the table while it is in set integrity pending state.
- A load replace operation takes place, or the NOT LOGGED INITIALLY WITH EMPTY TABLE option is activated, after the last integrity check on the table.
- A parent table is load replaced or checked for integrity non-incrementally.
- The table is in set integrity pending state before an upgrade. Full processing is required the first time the table is checked for integrity after an upgrade.
- The table space containing the table or its parent is rolled forward to a point in time and the table and its parent reside in different table spaces.

If a table has one or more `W` values in the CONST_CHECKED column of the SYSCAT.TABLES catalog, and if the NOT INCREMENTAL option is not specified in the SET INTEGRITY statement, the table is incrementally processed and the CONST_CHECKED column of SYSCAT.TABLES is marked as `U` to indicate that not all data has been verified by the system.

The SET INTEGRITY statement does not activate any DELETE triggers as a result of deleting rows that violate constraints, but once the table is removed from set integrity pending state, triggers are active. Thus, if you correct data and insert rows from the exception table into the loaded table, any INSERT triggers defined on the table are activated. The implications of this should be considered. One option is to drop the INSERT trigger, insert rows from the exception table, and then re-create the INSERT trigger.

## Statement dependencies when changing objects

Statement dependencies include package and cached dynamic SQL and XQuery statements. A *package* is a database object that contains the information needed by the database manager to access data in the most efficient way for a particular application program. *Binding* is the process that creates the package the database manager needs in order to access the database when the application is executed.

Packages and cached dynamic SQL and XQuery statements can be dependent on many types of objects.

These objects could be explicitly referenced, for example, a table or user-defined function that is involved in an SQL SELECT statement. The objects could also be implicitly referenced, for example, a dependent table that needs to be checked to ensure that **referential constraints** are not violated when a row in a parent table is deleted. Packages are also dependent on the privileges which have been granted to the package creator.

If a package or cached dynamic query statement depends on an object and that object is dropped, the package or cached dynamic query statement is placed in an "invalid" state. If a package depends on a user-defined function and that function is dropped, the package is placed in an "inoperative" state, with the following conditions:

- A cached dynamic SQL or XQuery statement that is in an invalid state is automatically re-optimized on its next use. If an object required by the statement has been dropped, execution of the dynamic SQL or XQuery statement might fail with an error message.
- A package that is in an invalid state is implicitly rebound on its next use. Such a package can also be explicitly rebound. If a package was marked as being not valid because a trigger was dropped, the rebound package no longer invokes the trigger.
- A package that is in an inoperative state must be explicitly rebound before it can be used.

Federated database objects have similar dependencies. For example, dropping a server or altering a server definition invalidates any packages or cached dynamic SQL referencing nicknames associated with that server.

In some cases, it is not possible to rebind the package. For example, if a table has been dropped and not re-created, the package cannot be rebound. In this case, you must either re-create the object or change the application so it does not use the dropped object.

In many other cases, for example if one of the **constraints** was dropped, it is possible to rebind the package.

The following system catalog views help you to determine the state of a package and the package's dependencies:

- SYSCAT.PACKAGEAUTH
- SYSCAT.PACKAGEDEP
- SYSCAT.PACKAGES

## Viewing constraint definitions for a table

Constraint definitions on a table can be found in the SYSCAT.INDEXES and SYSCAT.REFERENCES catalog views.

### About this task

The UNIQUERULE column of the SYSCAT.INDEXES view indicates the characteristic of the index. If the value of this column is P, the index is a primary key, and if it is U, the index is a unique index (but not a primary key).

The SYSCAT.REFERENCES catalog view contains referential integrity (foreign key) constraint information.

## Dropping constraints

You can explicitly drop a table check constraint using the ALTER TABLE statement, or implicitly drop it as the result of a DROP TABLE statement.

## About this task

To drop constraints, use the ALTER TABLE statement with the DROP or DROP CONSTRAINT clauses. This allows you to **BIND** and continue accessing the tables that contain the affected columns. The name of all unique constraints on a table can be found in the SYSCAT.INDEXES system catalog view.

## Procedure

- To explicitly drop unique constraints, use the DROP UNIQUE clause of the ALTER TABLE statement.

  The DROP UNIQUE clause of the ALTER TABLE statement drops the definition of the unique constraint *constraint-name* and all referential constraints that are dependent upon this unique constraint. The *constraint-name* must identify an existing unique constraint.

  ```
  ALTER TABLE table-name
    DROP UNIQUE constraint-name
  ```

  Dropping this unique constraint invalidates any packages or cached dynamic SQL that used the constraint.

- To drop primary key constraints, use the DROP PRIMARY KEY clause of the ALTER TABLE statement.

  The DROP PRIMARY KEY clause of the ALTER TABLE statement drops the definition of the primary key and all referential constraints that are dependent upon this primary key. The table must have a primary key. To drop a primary key using the command line, enter:

  ```
  ALTER TABLE table-name
    DROP PRIMARY KEY
  ```

- To drop (table) check constraints, use the DROP CHECK clause of the ALTER TABLE statement.

  When you drop a check constraint, all packages and cached dynamic statements with INSERT or UPDATE dependencies on the table are invalidated. The name of all check constraints on a table can be found in the SYSCAT.CHECKS catalog view. Before attempting to drop a table check constraint having a system-generated name, look for the name in the SYSCAT.CHECKS catalog view.

  The following statement drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the table. To drop a table check constraint using the command line:

  ```
  ALTER TABLE table_name
    DROP CHECK check_constraint_name
  ```

  Alternatively, you can use the ALTER TABLE statement with the DROP CONSTRAINT option.

- To drop foreign key (referential) constraints, use the DROP CONSTRAINT clause of the ALTER TABLE statement.

  The DROP CONSTRAINT clause of the ALTER TABLE statement drops the constraint *constraint-name*. The *constraint-name* must identify an existing foreign key constraint, primary key, or unique constraint defined on the table. To drop foreign keys using the command line, enter:

  ```
  ALTER TABLE table-name
    DROP FOREIGN KEY foreign_key_name
  ```

  When a foreign key constraint is dropped, packages or cached dynamic statements containing the following might be marked as invalid:

  – Statements that insert or update the table containing the foreign key

– Statements that update or delete the parent table.

## Example

The following examples use the DROP PRIMARY KEY and DROP FOREIGN KEY clauses in the ALTER TABLE statement to drop primary keys and foreign keys on a table:

```
ALTER TABLE EMP_ACT
  DROP PRIMARY KEY
  DROP FOREIGN KEY ACT_EMP_REF
  DROP FOREIGN KEY ACT_PROJ_REF
ALTER TABLE PROJECT
  DROP PRIMARY KEY
```

# Chapter 23. Views

A *view* is an efficient way of representing data without the need to maintain it. A view is not an actual table and requires no permanent storage. A "virtual table" is created and used.

A *view* provides a different way of looking at the data in one or more tables; it is a named specification of a result table. The specification is a SELECT statement that is run whenever the view is referenced in an SQL statement. A view has columns and rows just like a table. All views can be used just like tables for data retrieval. Whether a view can be used in an insert, update, or delete operation depends on its definition.

A view can include all or some of the columns or rows contained in the tables on which it is based. For example, you can join a department table and an employee table in a view, so that you can list all employees in a particular department.

Figure 17 shows the relationship between tables and views.

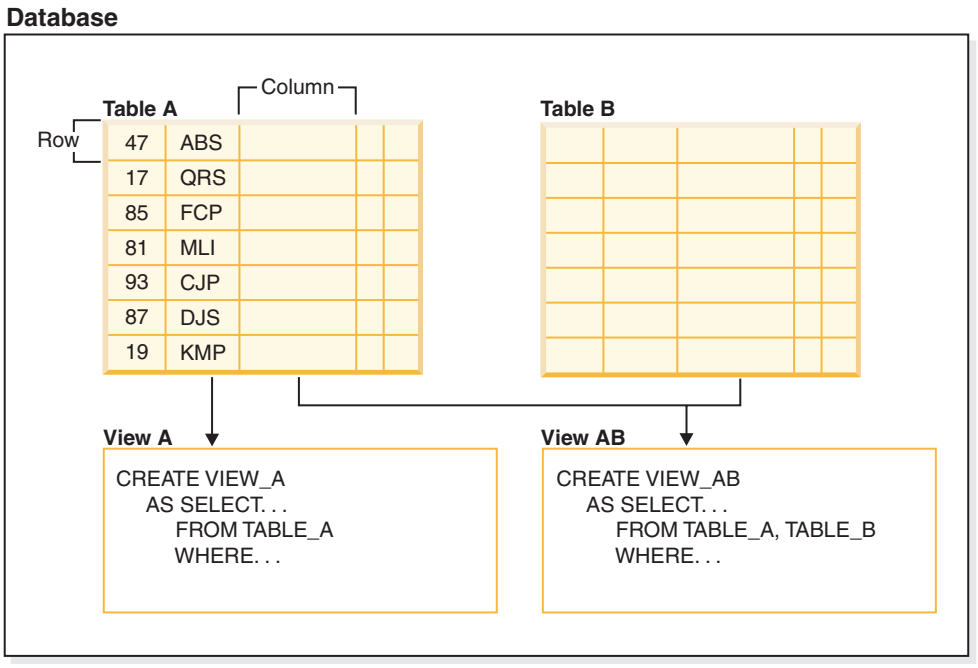



*Figure 17. Relationship between tables and views*

You can use views to control access to sensitive data, because views allow multiple users to see different presentations of the same data. For example, several users might be accessing a table of data about employees. A manager sees data about his or her employees but not employees in another department. A recruitment officer sees the hire dates of all employees, but not their salaries; a financial officer sees the salaries, but not the hire dates. Each of these users works with a view derived from the table. Each view appears to be a table and has its own name.

When the column of a view is directly derived from the column of a base table, that view column inherits any constraints that apply to the table column. For

example, if a view includes a foreign key of its table, insert and update operations using that view are subject to the same referential constraints as is the table. Also, if the table of a view is a parent table, delete and update operations using that view are subject to the same rules as are delete and update operations on the table.

A view can derive the data type of each column from the result table, or base the types on the attributes of a user-defined structured type. This is called a *typed view*. Similar to a typed table, a typed view can be part of a view hierarchy. A *subview* inherits columns from its *superview*. The term *subview* applies to a typed view and to all typed views that are below it in the view hierarchy. A *proper subview* of a view V is a view below V in the typed view hierarchy.

A view can become inoperative (for example, if the table is dropped); if this occurs, the view is no longer available for SQL operations.

## Views with the check option

A view that is defined WITH CHECK OPTION enforces any rows that are modified or inserted against the SELECT statement for that view. Views with the check option are also called *symmetric views*. For example, a symmetric view that only returns only employees in department 10 will not allow insertion of employees in other departments. This option, therefore, ensures the integrity of the data being modified in the database, returning an error if the condition is violated during an INSERT or UPDATE operation.

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

The WITH CHECK OPTION must not be specified for the following views:

- Views defined with the read-only option (a read-only view)
- View that reference the NODENUMBER or PARTITION function, a nondeterministic function (for example, RAND), or a function with external action
- Typed views

### Example 1

Following is an example of a view definition using the WITH CHECK OPTION. This option is required to ensure that the condition is always checked. The view ensures that the DEPT is always 10. This will restrict the input values for the DEPT column. When a view is used to insert a new value, the WITH CHECK OPTION is always enforced:

```
CREATE VIEW EMP_VIEW2
  (EMPNO, EMPNAME, DEPTNO, JOBTITLE, HIREDATE)
AS SELECT ID, NAME, DEPT, JOB, HIREDATE FROM EMPLOYEE
  WHERE DEPT=10
WITH CHECK OPTION;
```

If this view is used in an INSERT statement, the row will be rejected if the DEPTNO column is not the value 10. It is important to remember that there is no data validation during modification if the WITH CHECK OPTION is not specified.

If this view is used in a SELECT statement, the conditional (WHERE clause) would be invoked and the resulting table would only contain the matching rows of data. In other words, the WITH CHECK OPTION does not affect the result of a SELECT statement.

## Example 2

With a view, you can make a subset of table data available to an application program and validate data that is to be inserted or updated. A view can have column names that are different from the names of corresponding columns in the original tables. For example:

```
CREATE VIEW <name> (<column>, <column>, <column>)
   SELECT <column_name> FROM <table_name>
   WITH CHECK OPTION
```

## Example 3

The use of views provides flexibility in the way your programs and end-user queries can look at the table data.

The following SQL statement creates a view on the EMPLOYEE table that lists all employees in Department A00 with their employee and telephone numbers:

```
CREATE VIEW EMP_VIEW (DA00NAME, DA00NUM, PHONENO)
   AS SELECT LASTNAME, EMPNO, PHONENO FROM EMPLOYEE
   WHERE WORKDEPT = 'A00'
   WITH CHECK OPTION
```

The first line of this statement names the view and defines its columns. The name EMP_VIEW must be unique within its schema in SYSCAT.TABLES. The view name appears as a table name although it contains no data. The view will have three columns called DA00NAME, DA00NUM, and PHONENO, which correspond to the columns LASTNAME, EMPNO, and PHONENO from the EMPLOYEE table. The column names listed apply one-to-one to the select list of the SELECT statement. If column names are not specified, the view uses the same names as the columns of the result table of the SELECT statement.

The second line is a SELECT statement that describes which values are to be selected from the database. It might include the clauses ALL, DISTINCT, FROM, WHERE, GROUP BY, and HAVING. The name or names of the data objects from which to select columns for the view must follow the FROM clause.

## Example 4

The WITH CHECK OPTION clause indicates that any updated or inserted row to the view must be checked against the view definition, and rejected if it does not conform. This enhances data integrity but requires additional processing. If this clause is omitted, inserts and updates are not checked against the view definition.

The following SQL statement creates the same view on the EMPLOYEE table using the SELECT AS clause:

```
CREATE VIEW EMP_VIEW
   SELECT LASTNAME AS DA00NAME,
          EMPNO AS DA00NUM,
```

```
       PHONENO
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
WITH CHECK OPTION
```

For this example, the EMPLOYEE table might have salary information in it, which should not be made available to everyone. The employee's phone number, however, should be generally accessible. In this case, a view could be created from the LASTNAME and PHONENO columns only. Access to the view could be granted to PUBLIC, while access to the entire EMPLOYEE table could be restricted to those who have the authorization to see salary information.

## Creating views

Views are derived from one or more tables, nicknames, or views, and can be used interchangeably with tables when retrieving data. When changes are made to the data shown in a view, the data is changed in the table itself. The table, nickname, or view on which the view is to be based must already exist before the view can be created.

### About this task

A view can be created to limit access to sensitive data, while allowing more general access to other data.

When inserting into a view where the select list of the view definition directly or indirectly includes the name of an identity column of a table, the same rules apply as if the INSERT statement directly referenced the identity column of the table.

In addition to using views as described previously, a view can also be used to:

- Alter a table without affecting application programs. This can happen by creating a view based on an underlying table. Applications that use the underlying table are not affected by the creation of the new view. New applications can use the created view for different purposes than those applications that use the underlying table.
- Sum the values in a column, select the maximum values, or average the values.
- Provide access to information in one or more data sources. You can reference nicknames within the CREATE VIEW statement and create multi-location/global views (the view could join information in multiple data sources located on different systems).

  When you create a view that references nicknames using standard CREATE VIEW syntax, you will see a warning alerting you to the fact that the authentication ID of view users will be used to access the underlying object or objects at data sources instead of the view creator authentication ID. Use the FEDERATED keyword to suppress this warning.

A typed view is based on a predefined structured type. You can create a typed view using the CREATE VIEW statement.

An alternative to creating a view is to use a nested or common table expression to reduce catalog lookup and improve performance.

A sample CREATE VIEW statement is shown in the following example. The underlying table, EMPLOYEE, has columns named SALARY and COMM. For security reasons this view is created from the ID, NAME, DEPT, JOB, and

HIREDATE columns. In addition, access on the DEPT column is restricted. This definition will only show the information of employees who belong to the department whose DEPTNO is 10.

```
CREATE VIEW EMP_VIEW1
  (EMPID, EMPNAME, DEPTNO, JOBTITLE, HIREDATE)
  AS SELECT ID, NAME, DEPT, JOB, HIREDATE FROM EMPLOYEE
    WHERE DEPT=10;
```

After the view has been defined, the access privileges can be specified. This provides data security since a restricted view of the table is accessible. As shown in the previous example, a view can contain a WHERE clause to restrict access to certain rows or can contain a subset of the columns to restrict access to certain columns of data.

The column names in the view do not have to match the column names of the base table. The table name has an associated schema as does the view name.

Once the view has been defined, it can be used in statements such as SELECT, INSERT, UPDATE, and DELETE (with restrictions). The DBA can decide to provide a group of users with a higher level privilege on the view than the table.

## Dropping views

Use the DROP VIEW statement to drop views. Any views that are dependent on the view being dropped are made inoperative.

### Procedure

To drop a view by using the command line, enter:

```
DROP VIEW view_name
```

### Example

The following example shows how to drop a view named EMP_VIEW:

```
DROP VIEW EMP_VIEW
```

As in the case of a table hierarchy, it is possible to drop an entire view hierarchy in one statement by naming the root view of the hierarchy, as in the following example:

```
DROP VIEW HIERARCHY VPerson
```

# Chapter 24. Indexes

An *index* is a set of pointers that are logically ordered by the values of one or more keys. The pointers can refer to rows in a table, blocks in an MDC or ITC table, XML data in an XML storage object, and so on.

Indexes are used to:

- Improve performance. In most cases, access to data is faster with an index. Although an index cannot be created for a view, an index created for the table on which a view is based can sometimes improve the performance of operations on that view.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

As data is added to a table, it is appended to the bottom (unless other actions have been carried out on the table or the data being added). There is no inherent order to the data. When searching for a particular row of data, each row of the table from first to last must be checked. Indexes are used as a means to access the data within the table in an order that might otherwise not be available.

Typically, when you search for data in a table, you are looking for rows with columns that have specific values. A column value in a row of data can be used to identify the entire row. For example, an employee number would probably uniquely define a specific individual employee. Or, more than one column might be needed to identify the row. For example, a combination of customer name and telephone number. Columns in an index used to identify data rows are known as *keys*. A column can be used in more than one key.

An index is ordered by the values within a key. Keys can be unique or non-unique. Each table should have at least one unique key; but can also have other, non-unique keys. Each index has exactly one key. For example, you might use the employee ID number (unique) as the key for one index and the department number (non-unique) as the key for a different index.

Not all indexes point to rows in a table. MDC and ITC block indexes point to extents (or blocks) of the data. XML indexes for XML data use particular XML pattern expressions to index paths and values in XML documents stored within a single column. The data type of that column must be XML. Both MDC and ITC block indexes and XML indexes are system generated indexes.

## Example

Table A in Figure 18 on page 202 has an index based on the employee numbers in the table. This key value provides a pointer to the rows in the table. For example, employee number 19 points to employee KMP. An index allows efficient access to rows in a table by creating a path to the data through pointers.

Unique indexes can be created to ensure uniqueness of the index key. An *index key* is a column or an ordered collection of columns on which an index is defined. Using a unique index will ensure that the value of each index key in the indexed column or columns is unique.

Figure 18 shows the relationship between an index and a table.

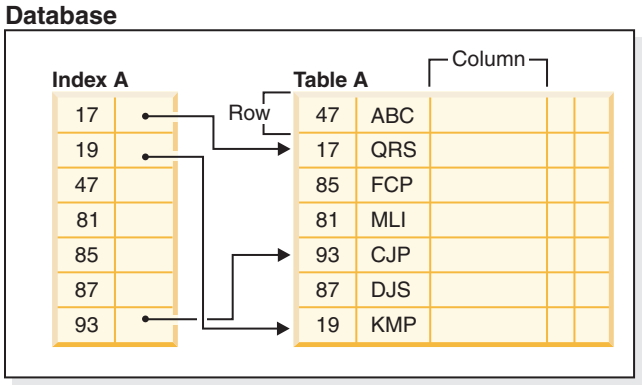



*Figure 18. Relationship between an index and a table*

Figure 19 illustrates the relationships among some database objects. It also shows that tables, indexes, and long data are stored in table spaces.

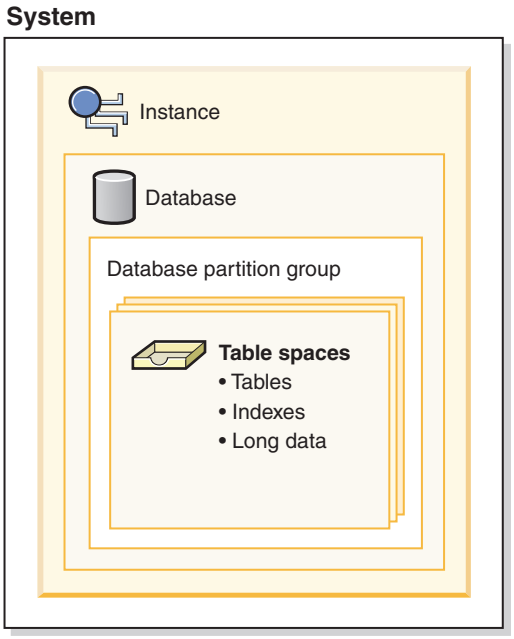



*Figure 19. Relationships among selected database objects*

## Types of indexes

There are different types of indexes that can be created for different purposes. For example, unique indexes enforce the constraint of uniqueness in your index keys; bidirectional indexes allow for scans in both the forward and reverse directions; clustered indexes can help improve the performance of queries that traverse the table in key order.

### Unique and non-unique indexes

Unique indexes are indexes that help maintain data integrity by ensuring that no two rows of data in a table have identical key values.

When attempting to create a unique index for a table that already contains data, values in the column or columns that comprise the index are checked for uniqueness; if the table contains rows with duplicate key values, the index creation process fails. Once a unique index has been defined for a table, uniqueness is enforced whenever keys are added or changed within the index. (This includes insert, update, load, import, and set integrity, to name a few.) In addition to enforcing the uniqueness of data values, a unique index can also be used to improve data retrieval performance during query processing.

Non-unique indexes, are not used to enforce constraints on the tables with which they are associated. Instead, non-unique indexes are used solely to improve query performance by maintaining a sorted order of data values that are used frequently.

## Clustered and non-clustered indexes

Index architectures are classified as clustered or non-clustered. Clustered indexes are indexes whose order of the rows in the data pages correspond to the order of the rows in the index. This is why only one clustered index can exist in a given table, whereas, many non-clustered indexes can exist in the table. In some relational database management systems, the leaf node of the clustered index corresponds to the actual data, not a pointer to data that resides elsewhere.

Both clustered and non-clustered indexes contain only keys and record IDs in the index structure. The record IDs always point to rows in the data pages. The only difference between clustered and non-clustered indexes is that the database manager attempts to keep the data in the data pages in the same order as the corresponding keys appear in the index pages. Thus the database manager will attempt to insert rows with similar keys onto the same pages. If the table is reorganized, it will be inserted into the data pages in the order of the index keys.

Reorganizing a table with respect to a chosen index re-clusters the data. A clustered index is most useful for columns that have range predicates because it allows better sequential access of data in the table. This results in fewer page fetches, since like values are on the same data page.

In general, only one of the indexes in a table can have a high degree of clustering.

Clustering indexes can improve the performance of most query operations because they provide a more linear access path to data, which has been stored in pages. In addition, because rows with similar index key values are stored together, sequential detection prefetching is usually more efficient when clustering indexes are used.

However, clustering indexes cannot be specified as part of the table definition used with the CREATE TABLE statement. Instead, clustering indexes are only created by executing the CREATE INDEX statement with the CLUSTER option specified. Then the ALTER TABLE statement should be used to add a primary key that corresponds to the clustering index created to the table. This clustering index will then be used as the table's primary key index.

**Note:** Setting PCTFREE in the table to an appropriate value using the ALTER TABLE statement can help the table remain clustered by leaving adequate free space to insert rows in the pages with similar values. For more information, see "ALTER TABLE statement" in the *SQL Reference* and "Reducing the need to reorganize tables and indexes" in *Troubleshooting and Tuning Database Performance*.

### Improving performance with clustering indexes

Generally, clustering is more effectively maintained if the clustering index is unique.

### Differences between primary key or unique key constraints and unique indexes

It is important to understand that there is no significant difference between a primary unique key constraint and a unique index. The database manager uses a combination of a unique index and the NOT NULL constraint to implement the relational database concept of primary and unique key constraints. Therefore, unique indexes do not enforce primary key constraints by themselves because they allow null values. (Although null values represent unknown values, when it comes to indexing, a null value is treated as being equal to other null values.)

Therefore, if a unique index consists of a single column, only one null value is allowed-more than one null value would violate the unique constraint. Similarly, if a unique index consists of multiple columns, a specific combination of values and nulls can be used only once.

### Bidirectional indexes

By default, bidirectional indexes allow scans in both the forward and reverse directions. The ALLOW REVERSE SCANS clause of the CREATE INDEX statement enables both forward and reverse index scans, that is, in the order defined at index creation time and in the opposite (or reverse) order. This option allows you to:

- Facilitate MIN and MAX functions
- Fetch previous keys
- Eliminate the need for the database manager to create a temporary table for the reverse scan
- Eliminate redundant reverse order indexes

If DISALLOW REVERSE SCANS is specified then the index cannot be scanned in reverse order. (But physically it will be exactly the same as an ALLOW REVERSE SCANS index.)

### Partitioned and nonpartitioned indexes

Partitioned data can have indexes that are nonpartitioned, existing in a single table space within a database partition, indexes that are themselves partitioned across one or more table spaces within a database partition, or a combination of the two. Partitioned indexes are particularly beneficial when performing roll-in operations with partitioned tables (attaching a data partition to another table using the ATTACH PARTITION clause on the ALTER table statement.)

## Clustering of nonpartitioned indexes on partitioned tables

Clustering indexes offer the same benefits for partitioned tables as they do for regular tables. However, care must be taken with the table partitioning key definitions when choosing a clustering index.

You can create a clustering index on a partitioned table using any clustering key. The database server attempts to use the clustering index to cluster data locally within each data partition. During a clustered insert operation, an index lookup is

performed to find a suitable record identifier (RID). This RID is used as a starting point in the table when looking for space in which to insert the record. To achieve optimal clustering with good performance, there should be a correlation between the index columns and the table partitioning key columns. One way to ensure such correlation is to prefix the index columns with the table partitioning key columns, as shown in the following example:

```
partition by range (month, region)
create index...(month, region, department) cluster
```

Although the database server does not enforce this correlation, there is an expectation that all keys in the index will be grouped together by partition IDs to achieve good clustering. For example, suppose that a table is partitioned on QUARTER and a clustering index is defined on DATE. There is a relationship between QUARTER and DATE, and optimal clustering of the data with good performance can be achieved because all keys of any data partition are grouped together within the index. Figure 20 on page 206 shows that optimal scan performance is achieved only when clustering correlates with the table partitioning key.

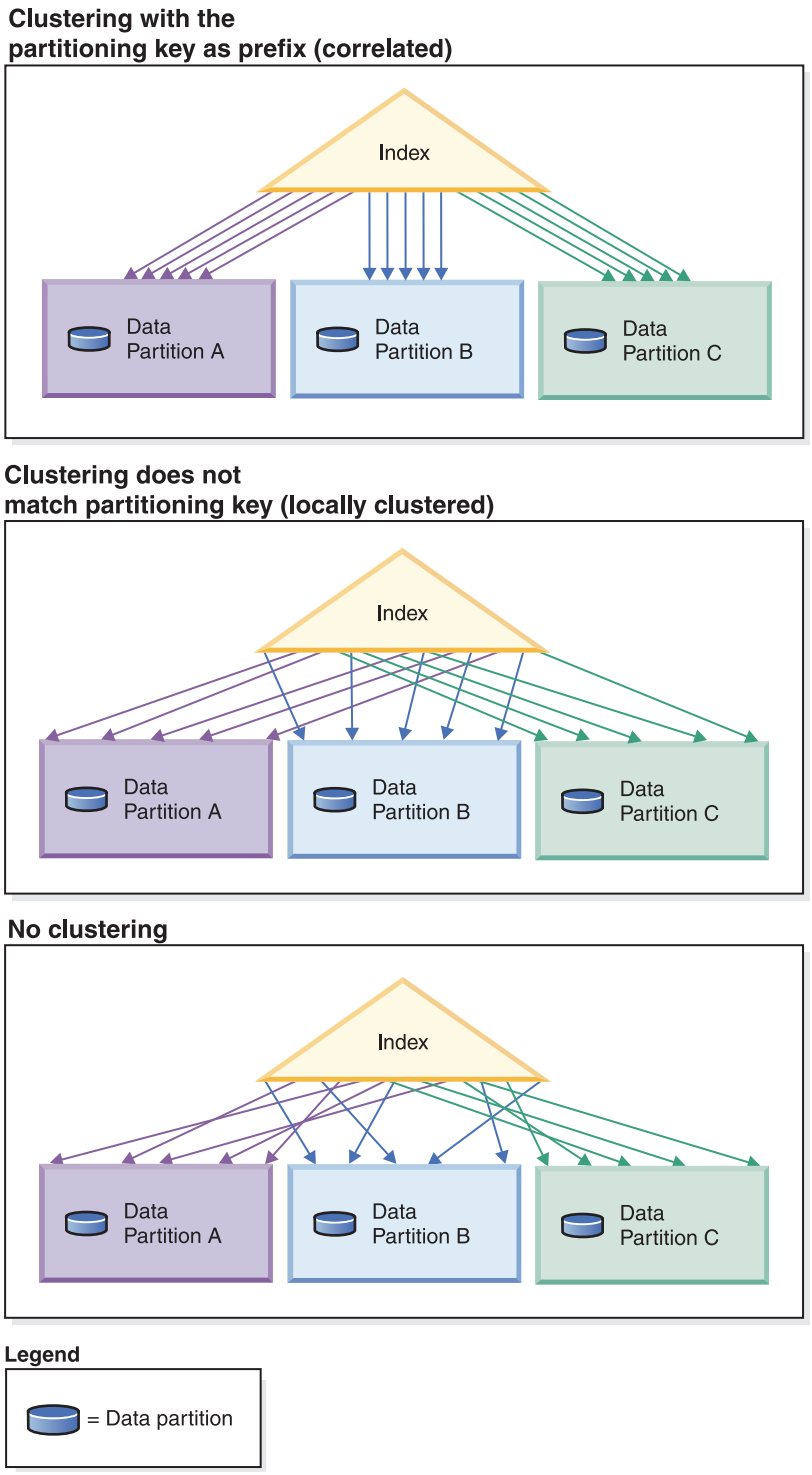



*Figure 20. The possible effects of a clustered index on a partitioned table.*

Benefits of clustering include:

- Rows are in key order within each data partition.
- Clustering indexes improve the performance of scans that traverse the table in key order, because the scanner fetches the first row of the first page, then each row in that same page before moving on to the next page. This means that only one page of the table needs to be in the buffer pool at any given time. In

contrast, if the table is not clustered, rows are likely fetched from different pages. Unless the buffer pool can hold the entire table, most pages will likely be fetched more than once, greatly slowing down the scan.

If the clustering key is not correlated with the table partitioning key, but the data is locally clustered, you can still achieve the full benefit of the clustered index if there is enough space in the buffer pool to hold one page of each data partition. This is because each fetched row from a particular data partition is near the row that was previously fetched from that same partition (see the second example in Figure 20 on page 206).

## Creating indexes

Indexes can be created for many reasons, including: to allow queries to run more efficiently; to order the rows of a table in ascending or descending sequence according to the values in a column; to enforce constraints such as uniqueness on index keys. You can use the CREATE INDEX statement, the DB2 Design Advisor , or the **db2advis** Design Advisor command to create the indexes.

### Before you begin

On Solaris platforms, patch 122300-11 on Solaris 9 or 125100-07 on Solaris 10 is required to create indexes with RAW devices. Without this patch, the CREATE INDEX statement hangs if a RAW device is used.

### About this task

This task assumes that you are creating an index on a nonpartitioned table.

### Procedure

To create an index from the command line, use the CREATE INDEX statement. For example:

```
CREATE UNIQUE INDEX EMP_IX
  ON EMPLOYEE(EMPNO)
  INCLUDE(FIRSTNAME, JOB)
```

The INCLUDE clause, applicable only on unique indexes, specifies additional columns to be appended to the set of index key columns. Any columns included with this clause are not used to enforce uniqueness. These included columns can improve the performance of some queries through index only access. This option might:

- Eliminate the need to access data pages for more queries
- Eliminate redundant indexes

If SELECT EMPNO, FIRSTNAME, JOB FROM EMPLOYEE is issued to the table on which this index resides, all of the required data can be retrieved from the index without reading data pages. This improves performance.

### What to do next

When a row is deleted or updated, the index keys are marked as deleted and are not physically removed from a page until cleanup is done some time after the deletion or update is committed. These keys are referred to as pseudo-deleted keys. Such a cleanup might be done by a subsequent transaction which is changing the page where the key is marked deleted. Clean up of pseudo-deleted keys can be

explicitly triggered by using the **CLEANUP ONLY ALL** parameter in the **REORG INDEXES** command.

# Dropping indexes

To delete an index, use the DROP statement.

## About this task

Other than changing the COMPRESSION attribute of an index, you cannot change any clause of an index definition; you must drop the index and create it again. Dropping an index does not cause any other objects to be dropped but might cause some packages to be invalidated.

Restrictions

A primary key or unique key index cannot be explicitly dropped. You must use one of the following methods to drop it:

- If the primary index or unique constraint was created automatically for the primary key or unique key, dropping the primary key or unique key causes the index to be dropped. Dropping is done through the ALTER TABLE statement.
- If the primary index or the unique constraint was user-defined, the primary key or unique key must be dropped first, through the ALTER TABLE statement. After the primary key or unique key is dropped, the index is no longer considered the primary index or unique index, and it can be explicitly dropped.

## Procedure

To drop an index by using the command line, enter:

```
DROP INDEX index_name
```

## Results

Any packages and cached dynamic SQL and XQuery statements that depend on the dropped indexes are marked invalid. The application program is not affected by changes resulting from adding or dropping indexes.

# Chapter 25. Triggers

A *trigger* defines a set of actions that are performed in response to an insert, update, or delete operation on a specified table. When such an SQL operation is executed, the trigger is said to have been *activated*. Triggers are optional and are defined using the CREATE TRIGGER statement.

Triggers can be used, along with referential constraints and check constraints, to enforce data integrity rules. Triggers can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts.

Triggers are a useful mechanism for defining and enforcing *transitional* business rules, which are rules that involve different states of the data (for example, a salary that cannot be increased by more than 10 percent).

Using triggers places the logic that enforces business rules inside the database. This means that applications are not responsible for enforcing these rules. Centralized logic that is enforced on all of the tables means easier maintenance, because changes to application programs are not required when the logic changes.

The following are specified when creating a trigger:

- The *subject table* specifies the table for which the trigger is defined.
- The *trigger event* defines a specific SQL operation that modifies the subject table. The event can be an insert, update, or delete operation.
- The *trigger activation time* specifies whether the trigger should be activated before or after the trigger event occurs.

The statement that causes a trigger to be activated includes a *set of affected rows*. These are the rows of the subject table that are being inserted, updated, or deleted. The *trigger granularity* specifies whether the actions of the trigger are performed once for the statement or once for each of the affected rows.

The *triggered action* consists of an optional search condition and a set of statements that are executed whenever the trigger is activated. The statements are only executed if the search condition evaluates to true. If the trigger activation time is before the trigger event, triggered actions can include statements that select, set transition variables, or signal SQL states. If the trigger activation time is after the trigger event, triggered actions can include statements that select, insert, update, delete, or signal SQL states.

The triggered action can refer to the values in the set of affected rows using *transition variables*. Transition variables use the names of the columns in the subject table, qualified by a specified name that identifies whether the reference is to the old value (before the update) or the new value (after the update). The new value can also be changed using the SET Variable statement in before, insert, or update triggers.

Another means of referring to the values in the set of affected rows is to use *transition tables*. Transition tables also use the names of the columns in the subject table, but specify a name to allow the complete set of affected rows to be treated as

a table. Transition tables can only be used in AFTER triggers (that is, not with BEFORE and INSTEAD OF triggers), and separate transition tables can be defined for old and new values.

Multiple triggers can be specified for a combination of table, event (INSERT, UPDATE, DELETE), or activation time (BEFORE, AFTER, INSTEAD OF). When more than one trigger exists for a particular table, event, and activation time, the order in which the triggers are activated is the same as the order in which they were created. Thus, the most recently created trigger is the last trigger to be activated.

The activation of a trigger might cause *trigger cascading*, which is the result of the activation of one trigger that executes statements that cause the activation of other triggers or even the same trigger again. The triggered actions might also cause updates resulting from the application of referential integrity rules for deletions that can, in turn, result in the activation of additional triggers. With trigger cascading, a chain of triggers and referential integrity delete rules can be activated, causing significant change to the database as a result of a single INSERT, UPDATE, or DELETE statement.

When multiple triggers have insert, update, or delete actions against the same object, conflict resolution mechanism, like temporary tables, are used to resolve access conflicts, and this can have a noticeable impact on performance, particularly in partitioned database environments.

## Types of triggers

A *trigger* defines a set of actions that are performed in response to an insert, update, or delete operation on a specified table. When such an SQL operation is executed, the trigger is said to have been *activated*. Triggers are optional and are defined using the CREATE TRIGGER statement.

Triggers can be used, along with referential constraints and check constraints, to enforce data integrity rules. Triggers can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts.

The following types of triggers are supported:

**BEFORE triggers**
Run before an update, or insert. Values that are being updated or inserted can be modified before the database is actually modified. You can use triggers that run before an update or insert in several ways:

- To check or modify values before they are actually updated or inserted in the database. This is useful if you must transform data from the way the user sees it to some internal database format.
- To run other non-database operations coded in user-defined functions.

**BEFORE DELETE triggers**
Run before a delete. Checks values (a raises an error, if necessary).

**AFTER triggers**
Run after an update, insert, or delete. You can use triggers that run after an update or insert in several ways:

- To update data in other tables. This capability is useful for maintaining relationships between data or in keeping audit trail information.

- To check against other data in the table or in other tables. This capability is useful to ensure data integrity when referential integrity constraints aren't appropriate, or when table check constraints limit checking to the current table only.
- To run non-database operations coded in user-defined functions. This capability is useful when issuing alerts or to update information outside the database.

**INSTEAD OF triggers**
Describe how to perform insert, update, and delete operations against views that are too complex to support these operations natively. They allow applications to use a view as the sole interface for all SQL operations (insert, delete, update and select).

## Designing triggers

When creating a trigger, you must associate it with a table; when creating an INSTEAD OF trigger, you must associate it with a view. This table or view is called the *target table* of the trigger. The term modify operation refers to any change in the state of the target table.

### About this task

A *modify operation* is initiated by:

- an INSERT statement
- an UPDATE statement, or a referential constraint which performs an UPDATE
- a DELETE statement, or a referential constraint which performs a DELETE
- a MERGE statement

You must associate each trigger with one of these three types of modify operations. The association is called the *trigger event* for that particular trigger.

You must also define the action, called the *triggered action*, that the trigger performs when its trigger event occurs. The triggered action consists of one or more statements which can execute either before or after the database manager performs the trigger event. Once a trigger event occurs, the database manager determines the set of rows in the subject table that the modify operation affects and executes the trigger.

**Guidelines when creating triggers:**
When creating a trigger, you must declare the following attributes and behavior:

- The name of the trigger.
- The name of the subject table.
- The trigger activation time (BEFORE or AFTER the modify operation executes).
- The trigger event (INSERT, DELETE, or UPDATE).
- The old transition variable value, if any.
- The new transition variable value, if any.
- The old transition table value, if any.
- The new transition table value, if any.
- The granularity (FOR EACH STATEMENT or FOR EACH ROW).

- The triggered action of the trigger (including a triggered action condition and triggered statement(s)).
- If the trigger event is UPDATE a trigger-column list if the trigger should only fire when specific columns are specified in the update statement.

**Designing multiple triggers:**

When triggers are defined using the CREATE TRIGGER statement, their creation time is registered in the database in form of a timestamp. The value of this timestamp is subsequently used to order the activation of triggers when there is more than one trigger that should be run at the same time. For example, the timestamp is used when there is more than one trigger on the same subject table with the same event and the same activation time. The timestamp is also used when there are one or more AFTER or INSTEAD OF triggers that are activated by the trigger event and referential constraint actions caused directly or indirectly (that is, recursively by other referential constraints) by the triggered action.

Consider the following two triggers:

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
  BEGIN ATOMIC
    UPDATE COMPANY_STATS
    SET NBEMP = NBEMP + 1;
  END

CREATE TRIGGER NEW_HIRED_DEPT
   AFTER INSERT ON EMPLOYEE
   REFERENCING NEW AS EMP
   FOR EACH ROW
     BEGIN ATOMIC
       UPDATE DEPTS
       SET NBEMP = NBEMP + 1
       WHERE DEPT_ID = EMP.DEPT_ID;
     END
```

The preceding triggers are activated when you run an INSERT operation on the employee table. In this case, the timestamp of their creation defines which of the preceding two triggers is activated first.

The activation of the triggers is conducted in ascending order of the timestamp value. Thus, a trigger that is newly added to a database runs after all the other triggers that are previously defined.

Old triggers are activated before new triggers to ensure that new triggers can be used as *incremental* additions to the changes that affect the database. For example, if a triggered statement of trigger T1 inserts a new row into a table T, a triggered statement of trigger T2 that is run after T1 can be used to update the same row in T with specific values. Because the activation order of triggers is predictable, you can have multiple triggers on a table and still know that the newer ones will be acting on a table that has already been modified by the older ones.

**Trigger interactions with referential constraints:**

A trigger event can occur as a result of changes due to referential constraint enforcement. For example, given two tables DEPT and EMP, if deleting or updating DEPT causes propagated deletes or updates to EMP by means of referential integrity constraints, then delete or update triggers defined on EMP become activated as a result of the referential constraint defined on DEPT. The triggers on EMP are run either BEFORE or AFTER

the deletion (in the case of ON DELETE CASCADE) or update of rows in EMP (in the case of ON DELETE SET NULL), depending on their activation time.

## Accessing old and new column values in triggers using transition variables

When you implement a FOR EACH ROW trigger, it might be necessary to refer to the value of columns of the row in the set of affected rows, for which the trigger is currently executing. Note that to refer to columns in tables in the database (including the subject table), you can use regular SELECT statements.

### About this task

A FOR EACH ROW trigger can refer to the columns of the row for which it is currently executing by using two transition variables that you can specify in the REFERENCING clause of a CREATE TRIGGER statement. There are two kinds of transition variables, which are specified as OLD and NEW, together with a correlation-name. They have the following semantics:

**OLD AS correlation-name**
Specifies a correlation name which captures the original state of the row, that is, before the triggered action is applied to the database.

**NEW AS correlation-name**
Specifies a correlation name which captures the value that is, or was, used to update the row in the database when the triggered action is applied to the database.

### Example

Consider the following example:

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED
  AND N_ROW.ORDER_PENDING = 'N')
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
    UPDATE PARTS SET PARTS.ORDER_PENDING = 'Y'
    WHERE PARTS.PARTNO = N_ROW.PARTNO;
  END
```

### What to do next

Based on the definition of the OLD and NEW transition variables given previously, it is clear that not every transition variable can be defined for every trigger. Transition variables can be defined depending on the kind of trigger event:

**UPDATE**
An UPDATE trigger can refer to both OLD and NEW transition variables.

**INSERT**
An INSERT trigger can only refer to a NEW transition variable because before the activation of the INSERT operation, the affected row does not

exist in the database. That is, there is no original state of the row that would define old values before the triggered action is applied to the database.

**DELETE**
A DELETE trigger can only refer to an OLD transition variable because there are no new values specified in the delete operation.

**Note:** Transition variables can only be specified for FOR EACH ROW triggers. In a FOR EACH STATEMENT trigger, a reference to a transition variable is not sufficient to specify to which of the several rows in the set of affected rows the transition variable is referring. Instead, refer to the set of new and old rows by using the OLD TABLE and NEW TABLE clauses of the CREATE TRIGGER statement. For more information about these clauses, see the CREATE TRIGGER statement.

## Creating triggers

A trigger defines a set of actions that are executed with, or triggered by, an INSERT, UPDATE, or DELETE clause on a specified table or a typed table.

### About this task

Use triggers to:

- Validate input data
- Generate a value for a newly inserted row
- Read from other tables for cross-referencing purposes
- Write to other tables for audit-trail purposes

You can use triggers to support general forms of integrity or business rules. For example, a trigger can check a customer's credit limit before an order is accepted or update a summary data table.

**Benefits:**

- Faster application development: Because a trigger is stored in the database, you do not have to code the actions that it performs in every application.
- Easier maintenance: After a trigger is defined, it is automatically invoked when the table that it is created on is accessed.
- Global enforcement of business rules: If a business policy changes, you only need to change the trigger and not each application program.

When creating an atomic trigger, care must be taken with the end-of-statement character. The command line processor, by default, considers a ";" the end-of-statement marker. You should manually edit the end-of-statement character in your script to create the atomic trigger so that you are using a character other than ";". For example, the ";" can be replaced by another special character like "#". You can also precede the CREATE TRIGGER DDL with:

```
--#SET TERMINATOR @
```

To change the terminator in the CLP on the fly, the following syntax sets it back:

```
--#SET TERMINATOR
```

To create a trigger from the command line, enter:

```
db2 -td delimiter -vf script
```

where the *delimiter* is the alternative end-of-statement character and the *script* is the modified script with the new *delimiter* in it.

A trigger body can include one or more of the following statements: INSERT, searched UPDATE, searched DELETE, fullselect, SET Variable, and SIGNAL SQLSTATE. The trigger can be activated before or after the INSERT, UPDATE, or DELETE statement to which it refers.

Restrictions

- You cannot use triggers with nicknames.
- If the trigger is a BEFORE trigger, the column name specified by the triggered action must not be a generated column other than an identity column. That is, the generated identity value is visible to BEFORE triggers.

### Procedure

To create a trigger from the command line, enter:

```
CREATE TRIGGER name
   action ON table_name
   operation
   triggered_action
```

### Example

The following statement creates a trigger that increases the number of employees each time a new person is hired, by adding 1 to the number of employees (NBEMP) column in the COMPANY_STATS table each time a row is added to the EMPLOYEE table.

```
CREATE TRIGGER NEW_HIRED
   AFTER INSERT ON EMPLOYEE
   FOR EACH ROW
   UPDATE COMPANY_STATS SET NBEMP = NBEMP+1;
```

## Modifying and dropping triggers

Triggers cannot be modified. They must be dropped and then created again according to the new definitions you require.

### Before you begin

**Trigger dependencies**

- All dependencies of a trigger on some other object are recorded in the SYSCAT.TRIGDEP system catalog view. A trigger can depend on many objects.
- If an object that a trigger is dependent on is dropped, the trigger becomes inoperative but its definition is retained in the system catalog view. To re-validate this trigger, you must retrieve its definition from the system catalog view and submit a new CREATE TRIGGER statement.
- If a trigger is dropped, its description is deleted from the SYSCAT.TRIGGERS system catalog view and all of its dependencies are deleted from the SYSCAT.TRIGDEP system catalog view. All packages having UPDATE, INSERT, or DELETE dependencies on the trigger are invalidated.
- If the view is dependent on the trigger and it is made inoperative, the trigger is also marked inoperative. Any packages dependent on triggers that have been marked inoperative are invalidated.

## About this task

A trigger object can be dropped using the DROP TRIGGER statement, but this procedure will cause dependent packages to be marked invalid, as follows:

- If an update trigger without an explicit column list is dropped, then packages with an update usage on the target table are invalidated.
- If an update trigger with a column list is dropped, then packages with update usage on the target table are only invalidated if the package also had an update usage on at least one column in the column-name list of the CREATE TRIGGER statement.
- If an insert trigger is dropped, packages that have an insert usage on the target table are invalidated.
- If a delete trigger is dropped, packages that have a delete usage on the target table are invalidated.

A package remains invalid until the application program is explicitly bound or rebound, or it is run and the database manager automatically rebinds it.

# Chapter 26. Sequences

A *sequence* is a database object that allows the automatic generation of values, such as cheque numbers. Sequences are ideally suited to the task of generating unique key values. Applications can use sequences to avoid possible concurrency and performance problems resulting from column values used to track numbers. The advantage that sequences have over numbers created outside the database is that the database server keeps track of the numbers generated. A crash and restart will not cause duplicate numbers from being generated.

The sequence numbers generated have the following properties:

- Values can be any exact numeric data type with a scale of zero. Such data types include: SMALLINT, BIGINT, INTEGER, and DECIMAL.
- Consecutive values can differ by any specified integer increment. The default increment value is 1.
- Counter value is recoverable. The counter value is reconstructed from logs when recovery is required.
- Values can be cached to improve performance. Pre-allocating and storing values in the cache reduces synchronous I/O to the log when values are generated for the sequence. In the event of a system failure, all cached values that have not been used are considered lost. The value specified for CACHE is the maximum number of sequence values that could be lost.

There are two expressions that can used with sequences:

- **NEXT VALUE expression:** returns the next value for the specified sequence. A new sequence number is generated when a NEXT VALUE expression specifies the name of the sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the counter for the sequence is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for each row of the result.
- **PREVIOUS VALUE expression:** returns the most recently generated value for the specified sequence for a previous statement within the current application process. That is, for any given connection, the PREVIOUS VALUE remains constant even if another connection invokes NEXT VALUE.

For complete details and examples of these expressions, see “Sequence reference” in *SQL Reference Volume 1*.

## Creating sequences

To create sequences, use the CREATE SEQUENCE statement. Unlike an identity column attribute, a sequence is not tied to a particular table column nor is it bound to a unique table column and only accessible through that table column.

### About this task

There are several restrictions on where NEXT VALUE or PREVIOUS VALUE expressions can be used. A sequence can be created, or altered, so that it generates values in one of these ways:

- Increment or decrement monotonically (changing by a constant amount) without bound
- Increment or decrement monotonically to a user-defined limit and stop
- Increment or decrement monotonically to a user-defined limit and cycle back to the beginning and start again

**Note: Use caution when recovering databases that use sequences:** For sequence values that are used outside the database, for example sequence numbers used for bank checkes, if the database is recovered to a point in time before the database failure, then this could cause the generation of duplicate values for some sequences. To avoid possible duplicate values, databases that use sequence values outside the database should not be recovered to a prior point in time.

To create a sequence called `order_seq` using defaults for all the options, issue the following statement in an application program or through the use of dynamic SQL statements:

```
CREATE SEQUENCE order_seq
```

This sequence starts at 1 and increases by 1 with no upper limit.

This example could represent processing for a batch of bank checks starting from 101 to 200. The first order would have been from 1 to 100. The sequence starts at 101 and increase by 1 with an upper limit of 200. NOCYCLE is specified so that duplicate cheque numbers are not produced. The number associated with the CACHE parameter specifies the maximum number of sequence values that the database manager preallocates and keeps in memory.

```
CREATE SEQUENCE order_seq
   START WITH 101
   INCREMENT BY 1
   MAXVALUE 200
   NOCYCLE
   CACHE 25
```

For more information about these and other options, and authorization requirements, see the CREATE SEQUENCE statement.

## Dropping sequences

To delete a sequence, use the DROP statement.

### Before you begin

When dropping sequences, the authorization ID of the statement must have DBADM authority.

Restrictions

Sequences that are system-created for IDENTITY columns cannot be dropped by using the DROP SEQUENCE statement.

### Procedure

To drop a specific sequence, enter:

```
DROP SEQUENCE sequence_name
```

where the *sequence_name* is the name of the sequence to be dropped and includes the implicit or explicit schema name to exactly identify an existing sequence.

## Results

Once a sequence is dropped, all privileges on the sequence are also dropped.

# Chapter 27. Aliases

An *alias* is an alternative name for an object such as a module, table or another alias. It can be used to reference an object wherever that object can be referenced directly.

An alias cannot be used in all contexts; for example, it cannot be used in the check condition of a check constraint. An alias cannot reference a declared temporary table but it can reference a created temporary table.

Like other objects, an alias can be created, dropped, and have comments associated with it. Aliases can refer to other aliases in a process called *chaining* as long as there are no circular references. Aliases do not require any special authority or privilege to use them. Access to the object referred to by an alias, however, does require the authorization associated with that object.

If an alias is defined as a *public* alias, it can be referenced by its unqualified name without any impact from the current default schema name. It can also be referenced using the qualifier SYSPUBLIC.

*Synonym* is an alternative name for alias.

For more information, refer to "Aliases in identifiers" in the *SQL Reference Volume 1*.

## Creating database object aliases

An *alias* is an indirect method of referencing a table, nickname, or view, so that an SQL or XQuery statement can be independent of the qualified name of that table or view.

### About this task

Only the alias definition must be changed if the table or view name changes. An alias can be created on another alias. An alias can be used in a view or trigger definition and in any SQL or XQuery statement, except for table check-constraint definitions, in which an existing table or view name can be referenced.

An alias can be defined for a table, view, or alias that does not exist at the time of definition. However, it must exist when the SQL or XQuery statement containing the alias is compiled.

An alias name can be used wherever an existing table name can be used, and can refer to another alias if no circular or repetitive references are made along the chain of aliases.

The alias name cannot be the same as an existing table, view, or alias, and can only refer to a table within the same database. The name of a table or view used in a CREATE TABLE or CREATE VIEW statement cannot be the same as an alias name in the same schema.

You do not require special authority to create an alias, unless the alias is in a schema other than the one owned by your current authorization ID, in which case DBADM authority is required.

When an alias, or the object to which an alias refers, is dropped, all packages dependent on the alias are marked as being not valid and all views and triggers dependent on the alias are marked inoperative.

**Note:** DB2 for z/OS employs two distinct concepts of aliases: ALIAS and SYNONYM. These two concepts differ from DB2 for Linux, UNIX, and Windows as follows:

- ALIASes in DB2 for z/OS:
  - Require their creator to have special authority or privilege
  - Cannot reference other aliases
- SYNONYMs in DB2 for z/OS:
  - Can only be used by their creator
  - Are always unqualified
  - Are dropped when a referenced table is dropped
  - Do not share namespace with tables or views

### Procedure

To create an alias using the command line, enter:

```
CREATE ALIAS alias_name FOR table_name
```

The following SQL statement creates an alias WORKERS for the EMPLOYEE table:

```
CREATE ALIAS WORKERS FOR EMPLOYEE
```

The alias is replaced at statement compilation time by the table or view name. If the alias or alias chain cannot be resolved to a table or view name, an error results. For example, if WORKERS is an alias for EMPLOYEE, then at compilation time:

```
SELECT * FROM WORKERS
```

becomes in effect

```
SELECT * FROM EMPLOYEE
```

## Dropping aliases

When you drop an alias, its description is deleted from the catalog, any packages, and cached dynamic queries that reference the alias are invalidated. All views and triggers dependent on the alias are marked inoperative.

### Procedure

To drop aliases, from the command line, issue the DROP statement:

```
DROP ALIAS employee-alias
```

# Chapter 28. User-defined routines

DB2 database systems provide routines that capture the functionality of most commonly used arithmetic, string, and casting functions. However, DB2 database systems also allow you to create routines to encapsulate logic of your own. These routines are called user-defined routines.

You can create your own procedures, functions and methods in any of the supported implementation styles for the routine type. Generally the prefix 'user-defined' is not used when referring to procedures and methods. User-defined functions are also commonly called UDFs.

## User-defined routine creation

User-defined procedures, functions and methods are created in the database by executing the appropriate CREATE statement for the routine type. These routine creation statements include:

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE METHOD

The clauses specific to each of the CREATE statements define characteristics of the routine, such as the routine name, the number and type of routine arguments, and details about the routine logic. DB2 database systems use the information provided by the clauses to identify and run the routine when it is invoked. Upon successful execution of the CREATE statement for a routine, the routine is created in the database. The characteristics of the routine are stored in the DB2 catalog views that users can query. Executing the CREATE statement to create a routine is also referred to as defining a routine or registering a routine.

User-defined routine definitions are stored in the SYSTOOLS system catalog table schema.

## User-defined routine logic implementation

There are three implementation styles that can be used to specify the logic of a routine:

- Sourced: user-defined routines can be *sourced* from the logic of existing built-in routines.
- SQL: user-defined routines can be implemented using only SQL statements.
- External: user-defined routines can be implemented using one of a set of supported programming languages.

  When routines are created in a non-SQL programming language, the library or class built from the code is associated with the routine definition by the value specified in the EXTERNAL NAME clause. When the routine is invoked the library or class associated with the routine is run.

User-defined routines can include a variety of SQL statements, but not all SQL statements.

User-defined routines are strongly typed, but type handling and error-handling mechanisms must be developed or enhanced by routine developers.

After a database upgrade, it may be necessary to verify or update routine implementations.

In general, user-defined routines perform well, but not as well as built-in routines.

User-defined routines can invoke built-in routines and other user-defined routines implemented in any of the supported formats. This flexibility allows users to essentially have the freedom to build a complete library of routine modules that can be re-used.

In general, user-defined routines provide a means for extending the SQL language and for modularizing logic that will be re-used by multiple queries or database applications where built-in routines do not exist.

## External routines

External routines are routines that have their logic implemented in a programming language application that resides outside of the database, in the file system of the database server.

The association of the routine with the external code application is asserted by the specification of the EXTERNAL clause in the CREATE statement of the routine.

You can create external procedures, external functions, and external methods. Although they are all implemented in external programming languages, each routine functional type has different features. Before deciding to implement an external routine, it is important that you first understand what external routines are, and how they are implemented and used, by reading the topic, "Overview of external routines". With that knowledge you can then learn more about external routines from the topics targeted by the related links so that you can make informed decisions about when and how to use them in your database environment.

### Supported routine programming languages

In general, routines are used to improve overall performance of the database management system by enabling application functionality to be performed on the database server. The amount of gain realized by these efforts is limited, to some degree, by the language chosen to write a routine.

Some of the issues you should consider before implementing routines in a certain language are:

- The available skills for developing a routine in a particular language and environment.
- The reliability and safety of a language's implemented code.
- The scalability of routines written in a particular language.

To help assess the preceding criteria, here are some characteristics of various supported languages:

**SQL**

- SQL routines are faster than Java routines, and roughly equivalent in performance to NOT FENCED C/C++ routines.
- SQL routines are written completely in SQL, and can include elements of SQL Procedural Language (SQL PL), which contains SQL control-statements that can be used to implement logic.

- SQL routines are considered 'safe' by DB2 database systems, as they consist entirely of SQL statements. SQL routines always run directly in the database engine, giving them good performance, and scalability.

**C/C++**

- Both C/C++ embedded SQL and DB2 CLI routines are faster than Java routines. They are roughly equivalent in performance to SQL routines when run in NOT FENCED mode.
- C/C++ routines are prone to error. It is recommended that you register C/C++ routines as FENCED NOT THREADSAFE, because routines in these languages are the most likely to disrupt the functioning of DB2's database engine by causing memory corruption. Running in FENCED NOT THREADSAFE mode, while safer, incurs performance overhead.

  For information on assessing and mitigating the risks of registering C/C++ routines as NOT FENCED or FENCED THREADSAFE, see the topic, "Security considerations for routines".
- By default, C/C++ routines run in FENCED NOT THREADSAFE mode to isolate them from damaging the execution of other routines. Because of this, you will have one `db2fmp` process per concurrently executing C/C++ routine on the database server. This can result in scalability problems on some systems.

**Java**

- Java routines are slower than C/C++ or SQL routines.
- Java routines are safer than C/C++ routines because control of dangerous operations is handled by the JVM. Because of this, reliability is increased, as it is difficult for a Java routine to damage another routine running in the same process.

  **Note:** To avoid potentially dangerous operations, Java Native Interface (JNI) calls from Java routines are not permitted. If you need to invoke C/C++ code from a Java routine, you can do so by invoking a separately cataloged C/C++ routine.
- When run in FENCED THREADSAFE mode (the default), Java routines scale well. All FENCED Java routines will share a few JVMs (more than one JVM might be in use on the system if the Java heap of a particular **`db2fmp`** process is approaching exhaustion).
- NOT FENCED Java routines are currently not supported. A Java routine defined as NOT FENCED will be invoked as if it had been defined as FENCED THREADSAFE.

**.NET common language runtime languages**

- .NET common language runtime (CLR) routines are routines that are compiled into intermediate language (IL) byte code that can be interpreted by the CLR of the .NET Framework. The source code for a CLR routine can be written in any .NET Framework supported language.
- Working with .NET CLR routines allows the user the flexibility to code in the .NET CLR supported programming language of their choice.
- CLR assemblies can be built up from sub-assemblies that were compiled from different .NET programming language source code, which allows the user to re-use and integrate code modules written in various languages.

- CLR routines can only be created as FENCED NOT THREADSAFE routines. This minimizes the possibility of engine corruption, but also means that these routines cannot benefit from the performance opportunity that can be had with NOT FENCED routines.

**OLE**

- OLE routines can be implemented in Visual C++, Visual Basic and other languages supported by OLE.
- The speed of OLE automated routines depends on the language used to implement them. In general, they are slower than non-OLE C/C++ routines.
- OLE routines can only run in FENCED NOT THREADSAFE mode. This minimizes the chance of engine corruption. This also means that OLE automated routines do not scale well.

**OLE DB**

- OLE DB can only be used to define table functions.
- OLE DB table functions connect to a external OLE DB data source.
- Depending on the OLE DB provider, OLE DB table functions are generally faster than Java table functions, but slower than C/C++ or SQL-bodied table functions. However, some predicates from the query where the function is invoked might be evaluated at the OLE DB provider, therefore reducing the number of rows that the DB2 database system has to process. This frequently results in improved performance.
- OLE DB routines can only run in FENCED NOT THREADSAFE mode. This minimizes the chance of engine corruption. This also means that OLE DB automated table functions do not scale well.

## External routine parameter styles

External routine implementations must conform to a particular convention for the exchange of routine parameter values. These conventions are known as *parameter styles*.

An external routine parameter style is specified when the routine is created by specifying the PARAMETER STYLE clause. Parameter styles characterize the specification and order in which parameter values will be passed to the external routine implementation. They also specify what if any additional values will be passed to the external routine implementation. For example, some parameter styles specify that for each routine parameter value that an additional separate null-indicator value be passed to the routine implementation to provide information about the parameters nullability which cannot otherwise be easily determined with a native programming language data type.

The following table provides a list of the available parameter styles, the routine implementations that support each parameter style, the functional routine types that support each parameter style, and a description of the parameter style:

*Table 20. Parameter styles*

| Parameter style | Supported language | Supported routine type | Description |
|---|---|---|---|
| SQL [1] | • C/C++<br>• OLE<br>• .NET common language runtime languages<br>• COBOL [2] | • UDFs<br>• stored procedures<br>• methods | In addition to the parameters passed during invocation, the following arguments are passed to the routine in the following order:<br>• A null indicator for each parameter or result declared in the CREATE statement.<br>• The SQLSTATE to be returned to the DB2 database system.<br>• The qualified name of the routine.<br>• The specific name of the routine.<br>• The SQL diagnostic string to be returned to the DB2 database system.<br>Depending on options specified in the CREATE statement and the routine type, the following arguments can be passed to the routine in the following order:<br>• A buffer for the scratchpad.<br>• The call type of the routine.<br>• The dbinfo structure (contains information about the database). |
| DB2SQL [1] | • C/C++<br>• OLE<br>• .NET common language runtime languages<br>• COBOL | • stored procedures | In addition to the parameters passed during invocation, the following arguments are passed to the stored procedure in the following order:<br>• A vector containing a null indicator for each parameter on the CALL statement.<br>• The SQLSTATE to be returned to the DB2 database system.<br>• The qualified name of the stored procedure.<br>• The specific name of the stored procedure.<br>• The SQL diagnostic string to be returned to the DB2 database system.<br>If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. |
| JAVA | • Java | • UDFs<br>• stored procedures | PARAMETER STYLE JAVA routines use a parameter passing convention that conforms to the Java language and SQLJ Routines specification.<br>For stored procedures, INOUT and OUT parameters will be passed as single entry arrays to facilitate the returning of values. In addition to the IN, OUT, and INOUT parameters, Java method signatures for stored procedures include a parameter of type ResultSet[] for each result set specified in the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement.<br>For PARAMETER STYLE JAVA UDFs and methods, no additional arguments to those specified in the routine invocation are passed.<br>PARAMETER STYLE JAVA routines do not support the DBINFO or PROGRAM TYPE clauses. For UDFs, PARAMETER STYLE JAVA can only be specified when there are no structured data types specified as parameters and no structured type, CLOB, DBCLOB, or BLOB data types specified as return types (SQLSTATE 429B8). Also, PARAMETER STYLE JAVA UDFs do not support table functions, call types, or scratchpads. |

*Table 20. Parameter styles (continued)*

| Parameter style | Supported language | Supported routine type | Description |
|---|---|---|---|
| DB2GENERAL | • Java | • UDFs<br>• stored procedures<br>• methods | This type of routine will use a parameter passing convention that is defined for use with Java methods. Unless you are developing table UDFs, UDFs with scratchpads, or need access to the dbinfo structure, it is recommended that you use PARAMETER STYLE JAVA.<br><br>For PARAMETER STYLE DB2GENERAL routines, no additional arguments to those specified in the routine invocation are passed. |
| GENERAL | • C/C++<br>• .NET common language runtime languages<br>• COBOL | • stored procedures | A PARAMETER STYLE GENERAL stored procedure receives parameters from the CALL statement in the invoking application or routine. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure.<br><br>GENERAL is the equivalent of SIMPLE stored procedures for DB2 for z/OS. |
| GENERAL WITH NULLS | • C/C++<br>• .NET common language runtime languages<br>• COBOL | • stored procedures | A PARAMETER STYLE GENERAL WITH NULLS stored procedure receives parameters from the CALL statement in the invoking application or routine. Also included is a vector containing a null indicator for each parameter on the CALL statement. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure.<br><br>GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS stored procedures for DB2 for z/OS. |

**Note:**

1. For UDFs and methods, PARAMETER STYLE SQL is equivalent to PARAMETER STYLE DB2SQL.
2. COBOL can only be used to develop stored procedures.
3. .NET common language runtime methods are not supported.

## Creating external routines

External routines including procedures and functions are created in a similar way as routines with other implementations, however there are a few more steps required, because the routine implementation requires the coding, compilation, and deployment of source code.

### Before you begin

- The IBM Data Server Client must be installed.
- The database server must be running an operating system that supports the chosen implementation programming language compilers and development software.
- The required compilers and runtime support for the chosen programming language must be installed on the database server
- Authority to execute the CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statement.

## About this task

You would choose to implement an external routine if:

- You want to encapsulate complex logic into a routine that accesses the database or that performs an action outside of the database.
- You require the encapsulated logic to be invoked from any of: multiple applications, the CLP, another routine (procedure, function (UDF), or method), or a trigger.
- You are most comfortable coding this logic in a programming language rather than using SQL and SQL PL statements.
- You require the routine logic to perform operations external to the database such as writing or reading to a file on the database server, the running of another application, or logic that cannot be represented with SQL and SQL PL statements.

## Procedure

1. Code the routine logic in the chosen programming language.
   - For general information about external routines, routine features, and routine feature implementation, see the topics referenced in the Prerequisites section.
   - Use or import any required header files required to support the execution of SQL statements.
   - Declare variables and parameters correctly using programming language data types that map to DB2 SQL data types.
2. Parameters must be declared in accordance with the format required by the parameter style for the chosen programming language. For more on parameters and prototype declarations see:
   - "External routine parameter styles" on page 226
3. Build your code into a library or class file.
4. Copy the library or class file into the DB2 *function directory* on the database server. It is recommended that you store assemblies or libraries associated with DB2 routines in the function directory. To find out more about the function directory, see the EXTERNAL clause of either of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the assembly to another directory on the server if you wish, but to successfully invoke the routine you must note the fully qualified path name of your assembly as you will require it for the next step.
5. Execute either dynamically or statically the appropriate SQL language CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION.
   - Specify the `LANGUAGE` clause with the appropriate value for the chosen API or programming language. Examples include: CLR, C, JAVA.
   - Specify the `PARAMETER STYLE` clause with the name of the supported parameter style that was implemented in the routine code.
   - Specify the `EXTERNAL` clause with the name of the library, class, or assembly file to be associated with the routine using one of the following values:
     – the fully qualified path name of the routine library, class, or assembly file .
     – the relative path name of the routine library, class, or assembly file relative to the function directory.

     By default DB2 will look for the library, class, or assembly file by name in the function directory unless a fully qualified or relative path name for it is specified in the EXTERNAL clause.

- Specify DYNAMIC RESULT SETS with a numeric value if your routine is a procedure and it will return one or more result sets to the caller.
- Specify any other clauses required to characterize the routine.

### What to do next

To invoke your external routine, see Routine invocation

## SQL routines

SQL routines are routines that have logic implemented with only SQL statements, including SQL Procedural Language (SQL PL) statements.

SQL routines are characterized by having their routine-body logic contained within the CREATE statement that is used to create them. You can create SQL procedures, SQL functions, and SQL methods. Although they are all implemented in SQL, each routine functional type has different features.

Before deciding to implement a SQL routine, it is important that you first understand what SQL routines are, how they are implemented, and used by reading an "Overview of routines". With that knowledge you can then learn more about SQL routine from the following concept topics so that you can make informed decisions about when and how to use them in your database environment:

- SQL procedures
- SQL functions
- Tools for developing SQL routines
- SQL Procedural Language (SQL PL)
- Comparison of SQL PL and inline SQL PL
- SQL PL statements and features
- Supported inline SQL PL statements and features
- Determining when to use SQL procedures or SQL functions
- Restrictions on SQL routines

After having learned about SQL routines, you might want to do one of the following tasks:

- Develop SQL procedures
- Develop SQL functions
- Develop SQL methods

## Creating SQL procedures from the command line

Before you begin, the user must have the privileges required to execute the CREATE PROCEDURE statement for an SQL procedure.

### Before you begin

- The user must have the privileges required to execute the CREATE PROCEDURE statement for an SQL procedure.
- Privileges to execute all of the SQL statements included within the SQL-procedure-body of the procedure.
- Any database objects referenced in the CREATE PROCEDURE statement for the SQL procedure must exist prior to the execution of the statement.

### Procedure

- Select an alternate terminating character for the Command Line Processor (DB2 CLP) other than the default terminating character, which is a semicolon (';'), to use in the script that you will prepare in the next step.

  This is required so that the CLP can distinguish the end of SQL statements that appear within the body of a routine's CREATE statement from the end of the CREATE PROCEDURE statement itself. The semicolon character must be used to terminate SQL statements within the SQL routine body and the chosen alternate terminating character should be used to terminate the CREATE statement and any other SQL statements that you might contain within your CLP script.

  For example, in the following CREATE PROCEDURE statement, the 'at;' sign ('@') is used as the terminating character for a DB2 CLP script named `myCLPscript.db2`:

```
    CREATE PROCEDURE UPDATE_SALARY_IF
    (IN employee_number CHAR(6), IN rating SMALLINT)
    LANGUAGE SQL
    BEGIN
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE EXIT HANDLER FOR not_found
         SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

      IF (rating = 1)
        THEN UPDATE employee
          SET salary = salary * 1.10, bonus = 1000
          WHERE empno = employee_number;
      ELSEIF (rating = 2)
        THEN UPDATE employee
          SET salary = salary * 1.05, bonus = 500
          WHERE empno = employee_number;
      ELSE UPDATE employee
          SET salary = salary * 1.03, bonus = 0
          WHERE empno = employee_number;
      END IF;
    END
@
```

- Run the DB2 CLP script containing the CREATE PROCEDURE statement for the procedure from the command line, using the following CLP command:

  ```
  db2 -td terminating-character -vf CLP-script-name
  ```

  where *terminating-character* is the terminating character used in the CLP script file *CLP-script-name* that is to be run.

  The DB2 CLP option **-td** indicates that the CLP terminator default is to be reset with *terminating-character*. The **-vf** indicates that the CLP's optional verbose (**-v**) option is to be used, which will cause each SQL statement or command in the script to be displayed to the screen as it is run, along with any output that results from its execution. The **-f** option indicates that the target of the command is a file.

  To run the specific script shown in the first step, issue the following command from the system command prompt:

  ```
  db2 -td@ -vf myCLPscript.db2
  ```

## Procedures

Procedures, also called stored procedures, are database objects created by executing the CREATE PROCEDURE statement. Procedures can encapsulate logic and SQL statement and can serve as sub-routine extensions to client applications, routines, triggers, and dynamic compound statements.

Procedures are invoked by executing the CALL statement with a reference to a procedure. Procedures can take input, output, and input-output parameters, execute a wide variety of SQL statements, and return multiple result sets to the caller.

**Features**

- Enable the encapsulation of logic elements and SQL statements that formulate a particular subroutine module
- Can be called from client applications, other routines, triggers, and dynamic compound statements - from anywhere that the CALL statement can be executed.
- Return multiple result-sets
- Support the execution of a large set of SQL statements including SQL statements that read or modify table data in both single and multiple partition databases
- Parameter support for input, output, and input-output parameters
- Nested procedure calls and function invocations are supported
- Recursive calls to procedures are supported
- Savepoints and transaction control are supported within procedures

**Limitations**

- Procedures cannot be invoked from within SQL statements other than the CALL statement. As an alternative, functions can be used to express logic that transforms column values.
- Output parameter values and result sets of procedure calls cannot be directly used by another SQL statement. Application logic must be used to assign these to variables that can be used in subsequent SQL statements.
- Procedures cannot preserve state between invocations.

**Common uses**

- Standardization of application logic
  - If multiple applications must similarly access or modify the database, a procedure can provide a single interface for the logic. The procedure is then available for re-use. Should the interface need to change to accommodate a change in business logic, only the single procedure must be modified.
- Isolation of database operations from non-database logic within applications
  - Procedures facilitate the implementation of sub-routines that encapsulate the logic and database accesses associated with a particular task that can be reused in multiple instances. For example, an employee management application can encapsulate the database operations specific to the task of hiring an employee. Such a procedure might insert employee information into multiple tables, calculate the employee's weekly pay based on an input parameter, and return the weekly pay value as an output parameter. Another procedure could do statistical analysis of data in a table and return result sets that contain the results of the analysis.
- Simplification of the management of privileges for a group of SQL statements
  - By allowing a grouping of multiple SQL statements to be encapsulated into one named database object, procedures allow

database administrators to manage fewer privileges. Instead of having to grant the privileges required to execute each of the SQL statements in the routine, they must only manage the privilege to invoke the routine.

**Supported implementations**

- There are built-in procedures that are ready-to-use, or users can create user-defined procedures. The following user-defined implementations are supported for procedures:
  - SQL implementation
  - External implementation

## Functions

Functions are relationships between sets of input data values and a set of result values. They enable you to extend and customize SQL. Functions are invoked from within elements of SQL statements such as a select-list or a FROM clause.

There are four types of functions:

- Aggregate functions
- Scalar functions
- Row functions
- Table functions

**Aggregate functions**

Also called a column function, this type of function returns a scalar value that is the result of an evaluation over a set of like input values. The similar input values can, for example, be specified by a column within a table, or by tuples in a VALUES clause. This set of values is called the argument set. For example, the following query finds the total quantity of bolts that are in stock or on order by using the SUM aggregate function:

```
SELECT SUM (qinstock + qonorder)
FROM inventory
WHERE description LIKE '%Bolt%'
```

**Scalar functions**

A scalar function is a function that, for each set of one or more scalar parameters, returns a single scalar value. Examples of scalar functions include the LENGTH function, and the SUBSTR function. Scalar functions can also be created that do complex mathematical calculations on function input parameters. Scalar functions can be referenced anywhere that an expression is valid within an SQL statement, such as in a select-list, or in a FROM clause. The following example shows a query that references the built-in LENGTH scalar function:

```
SELECT lastname, LENGTH(lastname)
FROM employee
```

**Row functions**

A row function is a function that for each set of one or more scalar parameters returns a single row. Row functions can only be used as a transform function mapping attributes of a structured type into built-in data type values in a row.

**Table functions**

Table functions are functions that for a group of sets of one or more parameters, return a table to the SQL statement that references it. Table functions can only be referenced in the FROM clause of a SELECT

statement. The table that is returned by a table function can participate in joins, grouping operations, set operations such as UNION, and any operation that could be applied to a read-only view. The following example demonstrates an SQL table function that updates an inventory table and returns the result set of a query on the updated inventory table:

```
CREATE FUNCTION updateInv(itemNo VARCHAR(20), amount INTEGER)
   RETURNS TABLE (productName VARCHAR(20),
                  quantity INTEGER)
   LANGUAGE SQL
   MODIFIES SQL DATA
   BEGIN ATOMIC

     UPDATE Inventory as I
       SET quantity = quantity + amount
         WHERE I.itemID = itemNo;

     RETURN
       SELECT I.itemName, I.quantity
         FROM Inventory as I
           WHERE I.itemID = itemNo;
   END
```

Functions provide support for the following features:

- Functions are supported across the DB2 brand database products including, among others, DB2, DB2 for z/OS, and DB2 Database for System i
- Moderate support for SQL statement execution
- Parameter support for input parameters and scalar or aggregate function return values
- Efficient compilation of function logic into queries that reference functions
- External functions provide support for storing intermediate values between the individual function sub-invocations for each row or value

There are built-in functions that are ready-to-use, or users can create user-defined functions. Functions can be implemented as SQL functions or as external functions. SQL functions can be either compiled or inlined. Inlined functions perform faster than compiled functions, but can execute only a subset of the SQL PL language. See the CREATE FUNCTION statement for more information.

## Methods

Methods allow you to access structured type attributes as well as to define additional behaviors for structured types.

A structured type is a user-defined data type containing one or more named attributes, each of which has a data type. Attributes are properties that describe an instance of a type. A geometric shape, for example, might have attributes such as its list of Cartesian coordinates.

Methods are generally implemented for a structured type to represent operations on the attributes of the structured type. For a geometric shape a method might calculate the volume of the shape. Methods share all of the features of scalar functions.

**Features**

- Ability to access structured type attributes
- Ability to set structured type attributes

- Ability to create operations on structured type attributes and return a function value
- Sensitive to the dynamic type of the subject type

**Limitations**

- Can only return a scalar value
- Can only be used with structured types
- Cannot be invoked for typed tables

**Common uses**

- Create operations on structured types
- Encapsulate the structured type

**Supported implementations**

There are no built-in methods. Users can create user-defined methods for existing user-defined structured types. Methods can be implemented using one of the following implementations:

- SQL routines
- External routines in C, C++, Java, C# (using OLE API), or Visual Basic (using OLE API)

SQL methods are easy to implement, but are generally designed in conjunction with the design of a structured type. External methods provide greater support for flexible logic implementation and allow a user to develop method logic in their preferred programming language.

# Chapter 29. DB2 compatibility features

The DB2 product provides a number of features that reduce the time and complexity of enabling some applications that were written for relational database products other than the DB2 product to run on a DB2 system.

Some of these features, including the following ones, are enabled by default.

- Implicit casting (weak typing), which reduces the number of SQL statements that you must modify to enable applications to run on the DB2 product.
- New built-in scalar functions. For details, see Built-in functions (see *SQL Reference Volume 1*).
- Improvements to the TIMESTAMP_FORMAT and VARCHAR_FORMAT scalar functions. The TIMESTAMP_FORMAT function returns a timestamp for an input string, using a specified format. The VARCHAR_FORMAT function returns a string representation of an input expression that has been formatted according to a specified character template. TO_DATE and TO_TIMESTAMP are synonyms for TIMESTAMP_FORMAT, and TO_CHAR is a synonym for VARCHAR_FORMAT.
- The lifting of several SQL restrictions, resulting in more compatible syntax between products. For example, the use of correlation names in subqueries and table functions is now optional.
- Synonyms for syntax that is used by other database products. Examples are as follows:
  - UNIQUE is a synonym for DISTINCT in the column functions and the select list of a query.
  - MINUS is a synonym for the EXCEPT set operator
  - You can use *seqname*.NEXTVAL in place of the SQL standard syntax NEXT VALUE FOR *seqname*. You can also use *seqname*.CURRVAL in place of the SQL standard syntax PREVIOUS VALUE FOR *seqname*.
- Global variables, which you can use to easily map package variables, emulate @@nested, @@level or @errorlevel global variables, or pass information from DB2 applications to triggers, functions, or procedures.
- An ARRAY collection data type that you use to easily map to VARRAY constructs in SQL procedures.
- Increased identifier length limits.
- The pseudocolumn ROWID, which you can use to refer to the RID. An unqualified ROWID reference is equivalent to RID_BIT(), and a qualified ROWID reference, such as EMPLOYEE.ROWID, is equivalent to RID_BIT(EMPLOYEE).

You can optionally enable the following other features by setting the **DB2_COMPATIBILITY_VECTOR** registry variable. These features are disabled by default.

- An implementation of hierarchical queries using CONNECT BY PRIOR syntax.
- Support for outer joins using the outer join operator, which is the plus sign (+).
- Use of the DATE data type as TIMESTAMP(0), a combined date and time value.
- Syntax and semantics to support the NUMBER data type.
- Syntax and semantics to support the VARCHAR2 data type.

- The ROWNUM pseudocolumn, which is a synonym for ROW_NUMBER() OVER(). However, the ROWNUM pseudocolumn is allowed in the SELECT list and in the WHERE clause of the SELECT statement.
- A dummy table named DUAL, which provides a capability that is similar to that of the SYSIBM.SYSDUMMY1 table.
- Alternative semantics for the TRUNCATE statement, such that IMMEDIATE is an optional keyword that is assumed to be the default if not specified. An implicit commit operation is performed before the TRUNCATE statement executes if the TRUNCATE statement is not the first statement in the logical unit of work.
- Support for assigning the CHAR or GRAPHIC data type instead of the VARCHAR or VARGRAPHIC data type to character and graphic string constants whose byte lengths are less than or equal to 254.
- Use of collection methods to perform operations on arrays, such as FIRST, LAST, NEXT, and previous.
- Support for creating Oracle data dictionary-compatible views.
- Support for compiling and executing PL/SQL statements and other language elements.
- Support for making cursors insensitive to subsequent statements by materializing the cursors at OPEN time.
- Support for INOUT parameters in procedures that you define with defaults and can invoke without specifying the arguments for those parameters.

### Additional resources

For more information about compatibility features, see DB2 Viper 2 compatibility features.

For information about the IBM Migration Toolkit (MTK), see Migrate Now!.

For information about the DB2 Oracle database compatibility features, see Oracle to DB2 Conversion Guide: Compatibility Made Easy.

## DATE data type based on TIMESTAMP(0)

The DATE data type supports applications that use the Oracle DATE data type and expect that the DATE values include time information (for example, '2009-04-01-09.43.05').

### Enablement

You enable DATE as TIMESTAMP(0) support at the database level, before creating the database where you require the support. To enable the support, set the **DB2_COMPATIBILITY_VECTOR** registry variable to hexadecimal value 0x40 (bit position 7), and then stop and restart the instance to have the new setting take effect.

```
db2set DB2_COMPATIBILITY_VECTOR=40
db2stop
db2start
```

To take full advantage of the DB2 compatibility features for Oracle applications, the recommended setting for the DB2_COMPATIBILITY_VECTOR is ORA, which sets all of the compatibility bits.

After you create a database with DATE as TIMESTAMP(0) support enabled, the **date_compat** database configuration parameter is set to ON.

If you create a database with DATE as TIMESTAMP(0) support enabled, you cannot disable that support for that database, even if you reset the **DB2_COMPATIBILITY_VECTOR** registry variable. Similarly, if you create a database with DATE as TIMESTAMP(0) support disabled, you cannot enable that support for that database later, even by setting the **DB2_COMPATIBILITY_VECTOR** registry variable.

## Effects

The **date_compat** database configuration parameter indicates whether the DATE compatibility semantics associated with the TIMESTAMP(0) data type are applied to the connected database. The effects of setting **date_compat** to ON are as follows.

When the DATE data type is explicitly encountered in SQL statements, it is implicitly mapped to TIMESTAMP(0) in most cases. An exception is the specification of SQL DATE in the *xml-index-specification* clause of a CREATE INDEX statement. As a result of the implicit mapping, messages refer to the TIMESTAMP data type instead of DATE, and any operations that describe data types for columns or routines return TIMESTAMP instead of DATE.

Datetime literal support is changed as follows:

- The value of an explicit DATE literal is a TIMESTAMP(0) value in which the time portion is all zeros. For example, `DATE '2008-04-28'` represents the timestamp value '2008-04-28-00.00.00'.
- The database manager supports two additional formats for the string representation of a date, which correspond to 'DD-MON-YYYY' and 'DD-MON-RR'. Only English abbreviations of the month are supported. For example, '28-APR-2008' or '28-APR-08' can be used as string representations of a date, which represents the TIMESTAMP(0) value '2008-04-28-00.00.00'.

  Starting from Version 9.7 Fix Pack 6, the database manager also supports the following formats for the string representation of a date in English only:

  – 'DDMONYYYY' or 'DDMONRR'
  – 'DD-MONYYYY' or 'DD-MONRR'
  – 'DDMON-YYYY' or 'DDMON-RR'

  For example, the following strings all represent the TIMESTAMP(0) value '2008-04-28-00.00.00':

  – '28APR2008' or '28APR08'
  – '28-APR2008' or '28-APR08'
  – '28APR-2008' or '28APR-08'

  For a description of the format elements, see TIMESTAMP_FORMAT scalar function (see *SQL Reference Volume 1*).

The CURRENT_DATE (also known as CURRENT DATE) special register returns a TIMESTAMP(0) value that is the same as the CURRENT_TIMESTAMP(0) value.

When you add a numeric value to a TIMESTAMP value or subtract a numeric value from a TIMESTAMP value, it is assumed that the numeric value represents a number of days. The numeric value can have any numeric data type, and any fractional value is considered to be a fractional portion of a day. For example, `TIMESTAMP '2008-03-28 12:00:00' + 1.3` adds 1 day, 7 hours, and 12 minutes to the TIMESTAMP value, resulting in '2008-03-29 19:12:00'. If you are using

expressions for partial days, such as 1/24 (1 hour) or 1/24/60 (1 minute), ensure that the **number_compat** database configuration parameter is set to ON so that the division is performed using DECFLOAT arithmetic.

The results of some functions change:

- If you pass a string argument to the ADD_MONTHS scalar function, it returns a TIMESTAMP(0) value.
- The DATE scalar function returns a TIMESTAMP(0) value for all input types.
- If you pass a string argument to the LAST_DAY scalar function, it returns a TIMESTAMP(0) value.
- If you pass a DATE() argument to the ADD_MONTHS, LAST_DAY, NEXT_DAY, ROUND, or TRUNCATE scalar function, the function returns a TIMESTAMP(0) value.
- The adding of one date value to another returns TIMESTAMP(0) value.
- The subtracting of one timestamp value from another returns DECFLOAT(34), representing the difference as a number of days. Similarly, subtracting one date value from another returns DECFLOAT(34), that represents a number of days.
- The second parameter in the TIMESTAMPDIFF scalar function does not represent a timestamp duration. Rather it represents the difference between two timestamps as a number of days. The returned estimate may vary by a number of days. For example, if the number of months (interval 64) is requested for the difference between '2010-03-31-00.00.00.000000' and '2010-03-01-00.00.00.000000', the result is 1. This is because the difference between the timestamps is 30 days, and the assumption of 30 days in a month applies. The following table shows how the returned value is determined for each interval.

*Table 21. TIMESTAMPDIFF computations*

| **Result interval** | **Computation using the difference between two timestamps as a number of days** |
|---|---|
| Years | integer value of (days/365) |
| Quarters | integer value of (days/90) |
| Months | integer value of (days/30) |
| Weeks | integer value of (days/7) |
| Days | integer value of days |
| Hours | integer value of (days*24) |
| Minutes (the absolute value of the number of days must not exceed 1491308.0888888888888882) | integer value of (days*24*60) |
| Seconds (the absolute value of the number of days must be less than 24855.1348148148148148) | integer value of (days*24*60*60) |
| Microseconds (the absolute value of the number of days must be less than 0.02485513481481481) | integer value of (days*24*60*60*1000000) |

If you use the import or load utility to input data into a DATE column, you must use the timestampformat file type modifier instead of the dateformat file type modifier.

# NUMBER data type

The NUMBER data type supports applications that use the Oracle NUMBER data type.

## Enablement

You enable NUMBER support at the database level, before creating the database where you require the support. To enable the support, set the **DB2_COMPATIBILITY_VECTOR** registry variable to hexadecimal value 0x10 (bit position 5), and then stop and restart the instance to have the new setting take effect.

```
db2set DB2_COMPATIBILITY_VECTOR=10
db2stop
db2start
```

To take full advantage of the DB2 compatibility features for Oracle applications, the recommended setting for the DB2_COMPATIBILITY_VECTOR is ORA, which sets all of the compatibility bits.

When you create a database with NUMBER support enabled, the **number_compat** database configuration parameter is set to ON.

If you create a database with NUMBER support enabled, you cannot disable NUMBER support for that database, even if you reset the **DB2_COMPATIBILITY_VECTOR** registry variable. Similarly, if you create a database with NUMBER support disabled, you cannot enable NUMBER support for that database later, even by setting the **DB2_COMPATIBILITY_VECTOR** registry variable.

## Effects

The effects of setting the **number_compat** database configuration parameter to ON are as follows.

When the NUMBER data type is explicitly encountered in SQL statements, the data type is implicitly mapped as follows:

- If you specify NUMBER without precision and scale attributes, it is mapped to DECFLOAT(16).
- If you specify NUMBER(*p*), it is mapped to DECIMAL(*p*).
- If you specify NUMBER(*p*,*s*), it is mapped to DECIMAL(*p*,*s*).

The maximum supported precision is 31, and the scale must be a positive value that is no greater than the precision. Also, a result of the implicit mapping, messages refer to data types DECFLOAT and DECIMAL instead of NUMBER. In addition, any operations that describe data types for columns or routines return either DECIMAL or DECFLOAT instead of NUMBER.

**Tip:** The DECFLOAT(16) data type provides a lower maximum precision than that of the Oracle NUMBER data type. If you need more than 16 digits of precision for storing numbers in columns, then explicitly define those columns as DECFLOAT(34).

Numeric literal support is unchanged: the rules for integer, decimal, and floating-point constants continue to apply. These rules limit decimal literals to 31 digits and floating-point literals to the range of binary double-precision floating-point values. If necessary, you can use a string-to-DECFLOAT(34) cast, using the CAST specification or the DECFLOAT function, for values beyond the

range of DECIMAL or DOUBLE up to the range of DECFLOAT(34). There is currently no support for a numeric literal that ends in either D, representing 64-bit binary floating-point values, or F, representing 32-bit binary floating-point values. A numeric literal that includes an E has the data type of DOUBLE, which you can cast to REAL using the CAST specification or the cast function REAL

If you cast NUMBER data values to character strings, using either the CAST specification or the VARCHAR or CHAR scalar function, all leading zeros are stripped from the result.

The default data type that is used for a sequence value in the CREATE SEQUENCE statement is DECIMAL(27) instead of INTEGER.

All arithmetic operations and arithmetic or mathematical functions involving DECIMAL or DECFLOAT data types are effectively performed using decimal floating-point arithmetic and return a value with a data type of DECFLOAT(34). This type of performance also applies to arithmetic operations where both operands have a DECIMAL or DECFLOAT(16) data type, which differs from the description of decimal arithmetic in the "Expressions with arithmetic operators" section of Expressions (see *SQL Reference Volume 1*). Additionally, all division operations involving only integer data types (SMALLINT, INTEGER, or BIGINT) are effectively performed using decimal floating-point arithmetic. These operations return a value with a data type of DECFLOAT(34) instead of an integer data type. Division by zero with integer operands returns infinity and a warning instead of an error.

In some cases function resolution is also changed, such that an argument of data type DECIMAL is considered to be a DECFLOAT value during the resolution process. Also functions with arguments that correspond to the NUMBER(*p*[,*s*]) data type are effectively treated as if the argument data types were NUMBER. However, this change in function resolution does not apply to the set of functions that have a variable number of arguments and base their result data types on the set of data types of the arguments. The functions included in this set are as follows:

- COALESCE
- DECODE
- GREATEST
- LEAST
- MAX (scalar)
- MIN (scalar)
- NVL
- VALUE

The rules for result data types (see *SQL Reference Volume 1*) are extended to make DECFLOAT(34) the result data type if the precision of a DECIMAL result data type would have exceeded 31. These rules also apply to the following items:

- Corresponding columns in set operations: UNION, EXCEPT(MINUS), and INTERSECT
- Expression values in the IN list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause

The rounding mode that is used for assignments and casts depends on the data types that are involved. In some cases, truncation is used. In cases where the target is a binary floating-point (REAL or DOUBLE) value, round-half-even is used, as usual. In other cases, usually involving a DECIMAL or DECFLOAT value, the

rounding is based on the value of the **decflt_rounding** database configuration parameter. The value of this parameter defaults to round-half-even, but you can set it to round-half-up to match the Oracle rounding mode. The following table summarizes the rounding that is used for various numeric assignments and casts.

*Table 22. Rounding for numeric assignments and casts*

| | Target data type | | | |
|---|---|---|---|---|
| **Source data type** | **Integer types** | **DECIMAL** | **DECFLOAT** | **REAL/DOUBLE** |
| **Integer types** | not applicable | not applicable | decflt_rounding | round_half_even |
| **DECIMAL** | decflt_rounding | decflt_rounding | decflt_rounding | round_half_even |
| **DECFLOAT** | decflt_rounding | decflt_rounding | decflt_rounding | round_half_even |
| **REAL/DOUBLE** | truncate | decflt_rounding | decflt_rounding | round_half_even |
| **String (cast only)** | not applicable | decflt_rounding | decflt_rounding | round_half_even |

The DB2 decimal floating-point values are based on the IEEE 754R standard. Retrieval of DECFLOAT data and casting of DECFLOAT data to character strings removes any trailing zeros after the decimal point.

### Client-server compatibility

Client applications working with a DB2 database server that you enable for NUMBER data type support never receive a NUMBER data type from the server. Any column or expression that would report NUMBER from an Oracle server report either DECIMAL or DECFLOAT from a DB2 server.

Because an Oracle environment expects the rounding mode to be round-half-up, it is important that the client rounding mode match the server rounding mode. This means that the `db2cli.ini` file setting must match the value of the **decflt_rounding** database configuration parameter. To most closely match the Oracle rounding mode, you should specify `ROUND_HALF_UP` for the database configuration parameter.

### Restrictions

NUMBER data type support has the following restrictions:

- There is no support for the following items:
  - A precision attribute greater than 31
  - A precision attribute of asterisk (*)
  - A scale attribute that exceeds the precision attribute
  - A negative scale attribute

  There is no corresponding DECIMAL precision and scale support for NUMBER data type specifications.
- You cannot invoke the trigonometric functions or the DIGITS scalar function with arguments of data type NUMBER without a precision (DECFLOAT).
- You cannot create a distinct type with the name NUMBER.

## VARCHAR2 and NVARCHAR2 data types

The VARCHAR2 and NVARCHAR2 data types support applications that use the Oracle VARCHAR2 and NVARCHAR2 data types.

## Enablement

You enable VARCHAR2 and NVARCHAR2 (subsequently jointly referred to as VARCHAR2) support at the database level, before creating the database where you require support. To enable the support, set the **DB2_COMPATIBILITY_VECTOR** registry variable to hexadecimal value 0x20 (bit position 6), and then stop and restart the instance to have the new setting take effect.

```
db2set DB2_COMPATIBILITY_VECTOR=20
db2stop
db2start
```

To take full advantage of the DB2 compatibility features for Oracle applications, the recommended setting for the DB2_COMPATIBILITY_VECTOR is ORA, which sets all of the compatibility bits.

When you create a database with VARCHAR2 support enabled, the **varchar2_compat** database configuration parameter is set to ON.

If you create a database with VARCHAR2 support enabled, you cannot disable VARCHAR2 support for that database, even if you reset the **DB2_COMPATIBILITY_VECTOR** registry variable. Similarly, if you create a database with VARCHAR2 support disabled, you cannot enable VARCHAR2 support for that database later, even by setting the **DB2_COMPATIBILITY_VECTOR** registry variable.

To use the NVARCHAR2 data type, a database must be a Unicode database.

## Effects

The effects of setting the **varchar2_compat** database configuration parameter to ON are as follows.

When the VARCHAR2 data type is explicitly encountered in SQL statements, it is implicitly mapped to the VARCHAR data type. The maximum length for VARCHAR2 is the same as the maximum length for VARCHAR: 32, 672.Similarly, when the NVARCHAR2 data type is explicitly encountered in SQL statements, it is implicitly mapped to the VARGRAPHIC data type. The maximum length for NVARCHAR2 is the same as the maximum length for VARGRAPHIC: 16, 336.

Character string literals up to 254 bytes in length have a data type of CHAR. Character string literals longer than 254 bytes have a data type of VARCHAR.

Comparisons involving varying-length string types use non-padded comparison semantics, and comparisons with only fixed-length string types continue to use blank-padded comparison semantics, with two exceptions:

- Comparisons involving string column information from catalog views always use the IDENTITY collation with blank-padded comparison semantics, regardless of the database collation.
- String comparisons involving a data type with the FOR BIT DATA attribute always use the IDENTITY collation with blank-padded comparison semantics.

IN list expressions are treated as having a varying-length string data type if both of the following conditions are true:

- The result type for the IN list of an IN predicate would resolve to a fixed-length string data type
- The left operand of the IN predicate is a varying-length string data type

Character string values (other than LOB values) with a length of zero are generally treated as null values. An assignment or cast of an empty string value to CHAR, NCHAR, VARCHAR, or NVARCHAR produces a null value.

Functions that return character string arguments, or that are based on parameters with character string data types, also treat empty string CHAR, NCHAR, VARCHAR, or NVARCHAR values as null values. Special considerations apply for some functions when the **varchar2_compat** database configuration parameter is set to ON, as follows:

- CONCAT function and the concatenation operator. A null or empty string value is ignored in the concatenated result. The result type of the concatenation is shown in the following table.

*Table 23. Data Type and lengths of concatenated operands*

| Operands | Combined length attributes | Result |
|---|---|---|
| CHAR(A) CHAR(B) | <255 | CHAR(A+B) |
| CHAR(A) CHAR(B) | >254 | VARCHAR(A+B) |
| CHAR(A) VARCHAR(B) | - | VARCHAR(MIN(A+B,32672)) |
| VARCHAR(A) VARCHAR(B) | - | VARCHAR(MIN(A+B,32672)) |
| CLOB(A) CHAR(B) | - | CLOB(MIN(A+B, 2G)) |
| CLOB(A) VARCHAR(B) | - | CLOB(MIN(A+B, 2G)) |
| CLOB(A) CLOB(B) | | CLOB(MIN(A+B, 2G)) |
| GRAPHIC(A) GRAPHIC(B) | <128 | GRAPHIC(A+B) |
| GRAPHIC(A) GRAPHIC(B) | >128 | VARGRAPHIC(A+B) |
| GRAPHIC(A) VARGRAPHIC(B) | - | VARGRAPHIC(MIN(A+B,16336)) |
| VARGRAPHIC(A) VARGRAPHIC(B) | - | VARGRAPHIC(MIN(A+B,16336)) |
| DBCLOB(A) CHAR(B) | - | DBCLOB(MIN(A+B, 1G)) |
| DBCLOB(A) VARCHAR(B) | - | DBCLOB(MIN(A+B, 1G)) |
| DBCLOB(A) CLOB(B) | | DBCLOB(MIN(A+B, 1G)) |
| BLOB(A) BLOB(B) | - | BLOB(MIN(A+B, 2G)) |

- INSERT function. A null value or empty string as the fourth argument results in deletion of the number of bytes indicated by the third argument, beginning at the byte position indicated by the second argument from the first argument.
- LENGTH function. The value returned by the LENGTH function is the number of bytes in the character string. An empty string value returns the null value.
- REPLACE function. If all of the argument values have a data type of CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC, then:
  - A null value or empty string as the second argument is treated as an empty string, and consequently the first argument is returned as the result
  - A null value or empty string as the third argument is treated as an empty string, and nothing replaces the string that is removed from the source string by the second argument.

  If any argument value has a data type of CLOB or BLOB and any argument is the null value, the result is the null value. All three arguments of the REPLACE function must be specified.

- SUBSTR function. References to SUBSTR which have a character string input for the first argument will be replaced with an invocation to SUBSTRB. References to SUBSTR which have a national character (graphic) string input for the first argument will be replaced with an invocation to SUBSTR2.
- TRANSLATE function. The *from-string-exp* is the second argument, and the *to-string-exp* is the third argument. If the *to-string-exp* is shorter than the *from-string-exp*, the extra characters in the *from-string-exp* that are found in the *char-string-exp* (the first argument) are removed; that is, the default *pad-char* argument is effectively an empty string, unless a different pad character is specified in the fourth argument.
- TRIM function. If the trim character argument of a TRIM function invocation is a null value or an empty string, the function returns a null value.

In the ALTER TABLE statement or the CREATE TABLE statement, when a DEFAULT clause is specified without an explicit value for a column defined with the VARCHAR or the VARGRAPHIC data type, the default value is a blank character.

Empty strings are converted to a blank character when the database configuration parameter **varchar2_compat** is set to ON. For example:

- SYSCAT.DATAPARTITIONS.STATUS has a single blank character when the data partition is visible.
- SYSCAT.PACKAGES.PKGVERSION has a single blank character when the package version has not been explicitly set.
- SYSCAT.ROUTINES.COMPILE_OPTIONS has a null value when compile options have not been set.

If SQL statements use parameter markers, a data type conversion that affects VARCHAR2 usage can occur. For example, if the input value is a VARCHAR of length zero and it is converted to a LOB, the result will be a null value. However, if the input value is a LOB of length zero and it is converted to a LOB, the result will be a LOB of length zero. The data type of the input value can be affected by deferred prepare.

## Restrictions

The VARCHAR2 data type and associated character string processing support have the following restrictions:

- The VARCHAR2 length attribute qualifier CHAR is not accepted.
- The LONG VARCHAR and LONG VARGRAPHIC data types are not supported (but are not explicitly blocked) when the **varchar2_compat** database configuration parameter is set to ON.

# Part 4. Working with DB2 Data using SQL

Structured Query Language (SQL) is a standardized language for defining and manipulating data in a relational database.

In accordance with the relational model of data, the database is treated as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables.

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. The transformation occurs in two phases: preparation and binding.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish static SQL from dynamic SQL.

# Chapter 30. INSERT statement

The INSERT statement inserts rows into a table, nickname, or view, or the underlying tables, nicknames, or views of the specified fullselect.

Inserting a row into a nickname inserts the row into the data source object to which the nickname refers. Inserting a row into a view also inserts the row into the table on which the view is based, if no INSTEAD OF trigger is defined for the insert operation on this view. If such a trigger is defined, the trigger will be executed instead.

## Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

## Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- INSERT privilege on the target table, view, or nickname
- CONTROL privilege on the target table, view, or nickname
- DATAACCESS authority

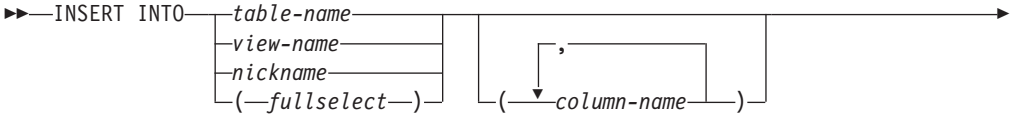
In addition, for each table, view, or nickname referenced in any fullselect used in the INSERT statement, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

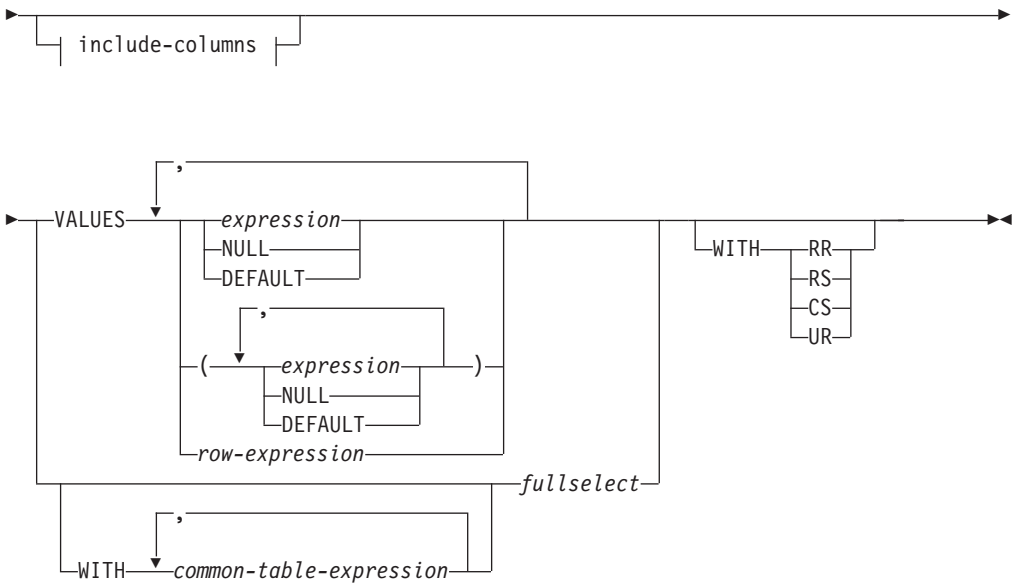
- SELECT privilege
- CONTROL privilege
- DATAACCESS authority

GROUP privileges are not checked for static INSERT statements.

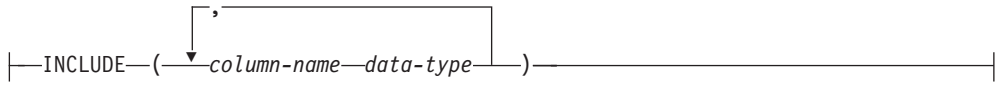
If the target of the insert operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

## Syntax






**include-columns:**




## Description

**INTO** *table-name*, *view-name*, *nickname*, **or** *(fullselect)*

Identifies the object of the insert operation. The name must identify one of the following objects:

- A table, view or nickname that exists at the application server
- A table or view at a remote server specified using a remote-object-name

The object must not be a catalog table, a system-maintained materialized query table, a view of a catalog table, or a read-only view, unless an INSTEAD OF trigger is defined for the insert operation on the subject view. Rows inserted into a nickname are placed in the data source object to which the nickname refers.

If the object of the insert operation is a fullselect, the fullselect must be insertable, as defined in the "Insertable views" Notes item in the description of the CREATE VIEW statement.

If the object of the insert operation is a nickname, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539).

If no INSTEAD OF trigger exists for the insert operation on this view, a value cannot be inserted into a view column that is derived from the following elements:

- A constant, expression, or scalar function
- The same base table column as some other column of the view

If the object of the insert operation is a view with such columns, a list of column names must be specified, and the list must not identify these columns.

A row can be inserted into a view or a fullselect that is defined using a UNION ALL if the row satisfies the check constraints of exactly one of the underlying base tables. If a row satisfies the check constraints of more than one table, or no table at all, an error is returned (SQLSTATE 23513).

A row cannot be inserted into a view or a fullselect that is defined using a UNION ALL if any base table of the view contains a before trigger and the before trigger contains an UPDATE, a DELETE, or an INSERT operation, or invokes any routine containing such operations (SQLSTATE 42987).

**(***column-name***,...)**
Specifies the columns for which insert values are provided. Each name must identify a column of the specified table, view, or nickname, or a column in the fullselect. The same column must not be identified more than once. If extended indicator variables are not enabled, a column that cannot accept inserted values (for example, a column based on an expression) must not be identified.

Omission of the column list is an implicit specification of a list in which every column of the table (that is not implicitly hidden) or view, or every item in the select-list of the fullselect is identified in left-to-right order. This list is established when the statement is prepared and, therefore, does not include columns that were added to a table after the statement was prepared.

*include-columns*
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the INSERT statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended at the end of the list of columns that are specified for *table-name* or *view-name*.

**INCLUDE**
Specifies a list of columns to be included in the intermediate result table of the INSERT statement. This clause can only be specified if the INSERT statement is nested in the FROM clause of a fullselect.

*column-name*
Specifies a column of the intermediate result table of the INSERT statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name* (SQLSTATE 42711).

*data-type*
Specifies the data type of the include column. The data type must be one that is supported by the CREATE TABLE statement.

**VALUES**
Introduces one or more rows of values to be inserted.

Each row specified in the VALUES clause must be assignable to the implicit or explicit column list and the columns identified in the INCLUDE clause, unless a row variable is used. When a row value list in parentheses is specified, the first value is inserted into the first column in the list, the second value into the second column, and so on. When a row expression is specified, the number of fields in the row type must match the number of names in the implicit or explicit column list.

*expression*
An *expression* can be any expression defined in the “Expressions” topic. If *expression* is a row type, it must not appear in parentheses. If *expression* is a variable, the host variable can include an indicator variable or in the case of a host structure, an indicator array, enabled for extended indicator variables. If extended indicator variables are enabled, the extended

indicator variable values of default (-5) or unassigned (-7) must not be used (SQLSTATE 22539) if either of the following statements is true:

- The expression is more complex than a single host variable with explicit casts
- The target column has data type of structured type

**NULL**
Specifies the null value and should only be specified for nullable columns.

**DEFAULT**
Specifies that the default value is to be used. The result of specifying DEFAULT depends on how the column was defined, as follows:

- If the column was defined as a generated column based on an expression, the column value is generated by the system, based on that expression.
- If the IDENTITY clause is used, the value is generated by the database manager.
- If the ROW CHANGE TIMESTAMP clause is used, the value for each inserted row is generated by the database manager as a timestamp that is unique for the table partition within the database partition.
- If the WITH DEFAULT clause is used, the value inserted is as defined for the column (see *default-clause* in "CREATE TABLE").
- If the NOT NULL clause is used and the GENERATED clause is not used, or the WITH DEFAULT clause is not used or DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).
- When inserting into a nickname, the DEFAULT keyword will be passed through the INSERT statement to the data source only if the data source supports the DEFAULT keyword in its query language syntax.

*row-expression*
Specifies any row expression of the type described in "Row expressions" that does not include a column name. The number of fields in the row must match the target of the insert and each field must be assignable to the corresponding column.

**WITH** *common-table-expression*
Defines a common table expression for use with the fullselect that follows.

*fullselect*
Specifies a set of new rows in the form of the result table of a fullselect. There may be one, more than one, or none. If the result table is empty, SQLCODE is set to +100 and SQLSTATE is set to '02000'.

When the base object of the INSERT and the base object of the fullselect or any subquery of the fullselect, are the same table, the fullselect is completely evaluated before any rows are inserted.

The number of columns in the result table must equal the number of names in the column list. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on.

If the expression that specifies the value of a result column is a variable, the host variable can include an indicator variable enabled for extended indicator variables. If extended indicator variables are enabled, and the expression is more than a single host variable, or a host variable being explicitly cast, then the extended indicator variable values of default or unassigned must not be

used (SQLSTATE 22539). The effects of default or unassigned values apply to the corresponding target columns of the *fullselect*.

**WITH**
Specifies the isolation level at which the fullselect is executed.

**RR** Repeatable Read

**RS** Read Stability

**CS** Cursor Stability

**UR** Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. The WITH clause has no effect on nicknames, which always use the default isolation level of the statement.

## Rules

- ***Triggers:*** INSERT statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the inserted values. If an insert operation into a view causes an INSTEAD OF trigger to fire, validity, referential integrity, and constraints will be checked against the updates that are performed in the trigger, and not against the view that caused the trigger to fire, or its underlying tables.
- ***Default values:*** The value inserted in any column that is not in the column list is either the default value of the column or null. Columns that do not allow null values and are not defined with NOT NULL WITH DEFAULT must be included in the column list. Similarly, if you insert into a view, the value inserted into any column of the base table that is not in the view is either the default value of the column or null. Hence, all columns of the base table that are not in the view must have either a default value or allow null values. The only value that can be inserted into a generated column defined with the GENERATED ALWAYS clause is DEFAULT (SQLSTATE 428C9).
- ***Length:*** If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number. If the insert value of a column is a string, the column must either be a string column with a length attribute at least as great as the length of the string, or a datetime column if the string represents a date, time, or timestamp.
- ***Assignment:*** Insert values are assigned to columns in accordance with specific assignment rules.
- ***Validity:*** If the table named, or the base table of the view named, has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes. If a view whose definition includes WITH CHECK OPTION is named, each row inserted into the view must conform to the definition of the view. For an explanation of the rules governing this situation, see "CREATE VIEW".
- ***Referential integrity:*** For each constraint defined on a table, each non-null insert value of the foreign key must be equal to a primary key value of the parent table.
- ***Check constraint:*** Insert values must satisfy the check conditions of the check constraints defined on the table. An INSERT to a table with check constraints defined has the constraint conditions evaluated once for each row that is inserted.
- ***XML values:*** A value that is inserted into an XML column must be a well-formed XML document (SQLSTATE 2200M).

- *Security policy:* If the identified table or the base table of the identified view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow:
  - Write access to all protected columns for which a data value is explicitly provided (SQLSTATE 42512)
  - Write access for any explicit value provided for a DB2SECURITYLABEL column for security policies that were created with the RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL option (SQLSTATE 23523)

  The session authorization ID must also have been granted a security label for write access for the security policy if an implicit value is used for a DB2SECURITYLABEL column (SQLSTATE 23523), which can happen when:
  - A value for the DB2SECURITYLABEL column is not explicitly provided
  - A value for the DB2SECURITYLABEL column is explicitly provided but the session authorization ID does not have write access for that value, and the security policy is created with the OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option
- **Extended indicator variable usage**: If enabled, negative indicator variable values outside the range of -1 through -7 must not be input (SQLSTATE 22010). Also, if enabled, the default and unassigned extended indicator variable values must not appear in contexts in which they are not supported (SQLSTATE 22539).
- **Extended indicator variables**: In an INSERT statement, a value of unassigned has the effect of setting the column to its default value.

  If the target column is a column defined as GENERATED ALWAYS, then it must be assigned the DEFAULT keyword, or the extended indicator variable-based values of default or unassigned (SQLSTATE 428C9).

## Examples

- *Example 1:* Insert a new department with the following specifications into the DEPARTMENT table:
  - Department number (DEPTNO) is 'E31'
  - Department name (DEPTNAME) is 'ARCHITECTURE'
  - Managed by (MGRNO) a person with number '00390'
  - Reports to (ADMRDEPT) department 'E01'.

```
INSERT INTO DEPARTMENT
  VALUES ('E31', 'ARCHITECTURE', '00390', 'E01')
```

- *Example 2:* Insert a new department into the DEPARTMENT table as in example 1, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT )
  VALUES ('E31', 'ARCHITECTURE', 'E01')
```

- *Example 3:* Insert two new departments using one statement into the DEPARTMENT table as in example 2, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('B11', 'PURCHASING', 'B01'),
         ('E41', 'DATABASE ADMINISTRATION', 'E01')
```

- *Example 4:* Create a temporary table MA_EMP_ACT with the same columns as the EMP_ACT table. Load MA_EMP_ACT with the rows from the EMP_ACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMP_ACT
        ( EMPNO CHAR(6)  NOT NULL,
          PROJNO CHAR(6)  NOT NULL,
          ACTNO SMALLINT  NOT NULL,
          EMPTIME DEC(5,2),
```

```
          EMSTDATE DATE,
          EMENDATE  DATE )
  INSERT INTO MA_EMP_ACT
    SELECT * FROM EMP_ACT
      WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

- *Example 5:* Use a C program statement to add a skeleton project to the PROJECT table. Obtain the project number (PROJNO), project name (PROJNAME), department number (DEPTNO), and responsible employee (RESPEMP) from host variables. Use the current date as the project start date (PRSTDATE). Assign a null value to the remaining columns in the table.

  ```
  EXEC SQL INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
    VALUES (:PRJNO, :PRJNM, :DPTNO, :REMP, CURRENT DATE);
  ```

- *Example 6:* Specify an INSERT statement as the *data-change-table-reference* within a SELECT statement. Define an extra include column whose values are specified in the VALUES clause, which is then used as an ordering column for the inserted rows.

  ```
  SELECT INORDER.ORDERNUM
    FROM NEW TABLE (INSERT INTO ORDERS(CUSTNO)INCLUDE (INSERTNUM INTEGER)
      VALUES(:CNUM1, 1), (:CNUM2, 2)) InsertedOrders
    ORDER BY INSERTNUM;
  ```

- *Example 7:* Use a C program statement to add a document to the DOCUMENTS table. Obtain values for the document ID (DOCID) column and the document data (XMLDOC) column from a host variable that binds to an SQL TYPE IS XML AS BLOB_FILE.

  ```
  EXEC SQL INSERT INTO DOCUMENTS
    (DOCID, XMLDOC) VALUES (:docid, :xmldoc)
  ```

- *Example 8:* For the following INSERT statements, assume that table SALARY_INFO is defined with three columns, and that the last column is an implicitly hidden ROW CHANGE TIMESTAMP column. In the following statement, the implicitly hidden column is explicitly referenced in the column list and a value is provided for it in the VALUES clause.

  ```
  INSERT INTO SALARY_INFO (LEVEL, SALARY, UPDATE_TIME)
    VALUES (2, 30000, CURRENT TIMESTAMP)
  ```

  The following INSERT statement uses an implicit column list. An implicit column list does not include implicitly hidden columns, so the VALUES clause only contains values for the other two columns.

  ```
  INSERT INTO SALARY_INFO VALUES (2, 30000)
  ```

  In this case, the UPDATE_TIME column must be defined to have a default value, and that default value is used for the row that is inserted.

# Chapter 31. UPDATE statement

The UPDATE statement updates the values of specified columns in rows of a table, view or nickname, or the underlying tables, nicknames, or views of the specified fullselect.

Updating a row of a view updates a row of its base table, if no INSTEAD OF trigger is defined for the update operation on this view. If such a trigger is defined, the trigger will be executed instead. Updating a row using a nickname updates a row in the data source object to which the nickname refers.

The forms of this statement are:

- The *Searched* UPDATE form is used to update one or more rows (optionally determined by a search condition).
- The *Positioned* UPDATE form is used to update exactly one row (as determined by the current position of a cursor).

## Invocation

An UPDATE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

## Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- UPDATE privilege on the target table, view, or nickname
- UPDATE privilege on each of the columns that are to be updated, including the columns of the BUSINESS_TIME period if a period-clause is specified
- CONTROL privilege on the target table, view, or nickname
- DATAACCESS authority

If a *row-fullselect* is included in the assignment, the privileges held by the authorization ID of the statement must include at least one of the following authorities for each referenced table, view, or nickname:

- SELECT privilege
- CONTROL privilege
- DATAACCESS authority

For each table, view, or nickname referenced by a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- SELECT privilege
- CONTROL privilege
- DATAACCESS authority

If the package used to process the statement is precompiled with SQL92 rules (option LANGLEVEL with a value of SQL92E or MIA), and the searched form of an UPDATE statement includes a reference to a column of the table, view, or

nickname in the right side of the *assignment-clause*, or anywhere in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

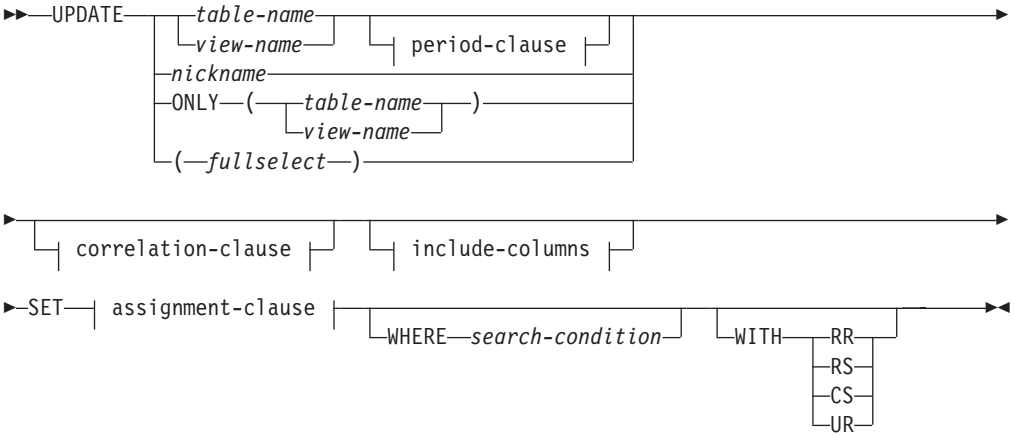
- SELECT privilege
- CONTROL privilege
- DATAACCESS authority

If the specified table or view is preceded by the ONLY keyword, the privileges held by the authorization ID of the statement must also include the SELECT privilege for every subtable or subview of the specified table or view.

GROUP privileges are not checked for static UPDATE statements.

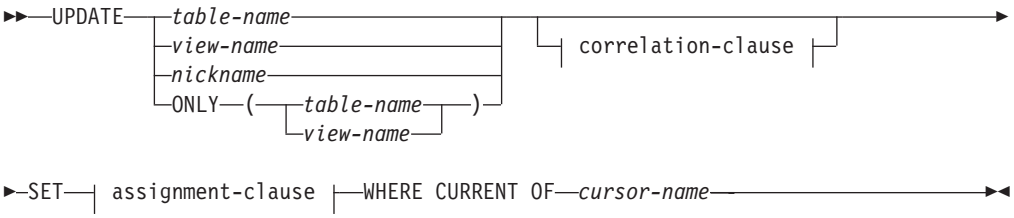
If the target of the update operation is a nickname, privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges that are required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

## Syntax (searched-update)

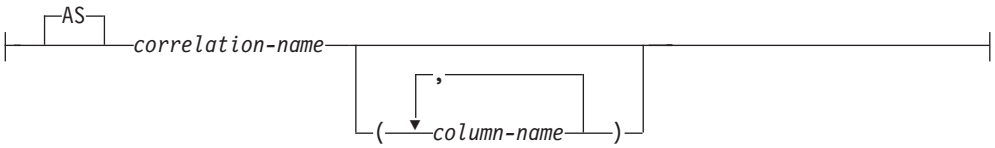



**period-clause:**

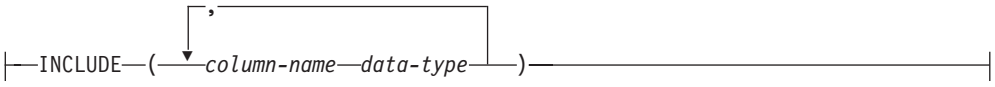

## Syntax (positioned-update)



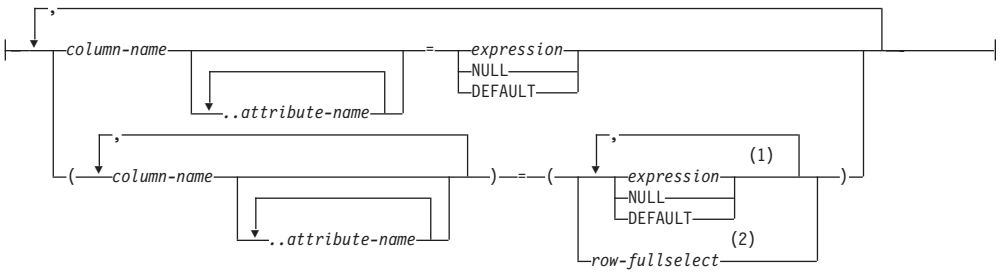
**correlation-clause:**




**include-columns:**




**assignment-clause:**




**Notes:**

1 The number of expressions, NULLs and DEFAULTs must match the number of column names.

2 The number of columns in the select list must match the number of column names.

## Description

*table-name*, *view-name*, *nickname*, **or** *(fullselect)*
Identifies the object of the update operation. The name must identify one of the following objects:

- A table, view, or nickname described in the catalog at the current server
- A table or view at a remote server specified using a remote-object-name

The object must not be a catalog table, a view of a catalog table (unless it is one of the updatable SYSSTAT views), a system-maintained materialized query table, or a read-only view that has no INSTEAD OF trigger defined for its update operations.

If *table-name* is a typed table, rows of the table or any of its proper subtables may get updated by the statement. Only the columns of the specified table may be set or referenced in the WHERE clause. For a positioned UPDATE, the associated cursor must also have specified the same table, view or nickname in the FROM clause without using ONLY.

If the object of the update operation is a fullselect, the fullselect must be updatable, as defined in the "Updatable views" Notes item in the description of the CREATE VIEW statement.

If the object of the update operation is a nickname, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539).

**ONLY (*table-name*)**
Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

**ONLY (*view-name*)**
Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

**period-clause**
Specifies that a period clause applies to the target of the update operation. If the target of the update operation is a view, the following conditions apply to the view:

- The FROM clause of the outer fullselect of the view definition must include a reference, directly or indirectly, to an application-period temporal table (SQLSTATE 42724M).
- An INSTEAD OF UPDATE trigger must not be defined for the view (SQLSTATE 428HY).

**FOR PORTION OF BUSINESS_TIME**
Specifies that the update only applies to row values for the portion of the period in the row that is specified by the period clause. The BUSINESS_TIME period must exist in the table (SQLSTATE 4274M).

**FROM *value1* TO *value2***
Specifies that the update applies to rows for the period specified from *value1* up to *value2*. No rows are updated if *value1* is greater than or equal to *value2*, or if *value1* or *value2* is the null value (SQLSTATE 02000).

For the period specified with FROM *value1* TO *value2*, the BUSINESS_TIME period in a row in the target of the update is in any of the following states:

- **Overlaps the beginning** of the specified period if the value of the begin column is less than *value1* and the value of the end column is greater than *value1*.
- **Overlaps the end** of the specified period if the value of the end column is greater than or equal to *value2* and the value of the begin column is less than *value2*.
- Is **fully contained** within the specified period if the value for the begin column for BUSINESS_TIME is greater than or equal to *value1* and the value for the corresponding end column is less than or equal to *value2*.
- Is **partially contained** in the specified period if the row overlaps the beginning of the specified period or the end of the specified period, but not both.

- **Fully overlaps** the specified period if the period in the row overlaps the beginning and end of the specified period.
- Is **not contained** in the period if both columns of BUSINESS_TIME are less than or equal to *value1* or greater than or equal to *value2*.

If the BUSINESS_TIME period in a row is not contained in the specified period, the row is not updated. Otherwise, the update is applied based on how the values in the columns of the BUSINESS_TIME period overlap the specified period as follows:

- If the BUSINESS_TIME period in a row is fully contained within the specified period, the row is updated and the values of the begin column and end column of BUSINESS_TIME are unchanged.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the beginning of the specified period:
  - The row is updated. In the updated row, the value of the begin column is set to *value1* and the value of the end column is the original value of the end column.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the end of the specified period:
  - The row is updated. In the updated row, the value of the begin column is the original value of the begin column and the end column is set to *value2*.
  - A row is inserted using the original values from the row, except that the begin column is set to *value2*.
- If the BUSINESS_TIME period in a row fully overlaps the specified period:
  - The row is updated. In the updated row the value of the begin column is set to *value1* and the value of the end column is set to *value2*.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
  - An additional row is inserted using the original values from the row, except that the begin column is set to *value2*.

*value1* **and** *value2*

Each expression must return a value that has a date data type, timestamp data type, or a valid data type for a string representation of a date or timestamp (SQLSTATE 428HY). The result of each expression must be comparable to the data type of the columns of the specified period (SQLSTATE 42884). See the comparison rules described in "Assignments and comparisons".

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable
- Scalar function whose arguments are supported operands (though user-defined functions and non-deterministic functions cannot be used)

- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

**correlation-clause**
Can be used within *search-condition* or *assignment-clause* to designate a table, view, nickname, or fullselect. For a description of *correlation-clause*, see "table-reference" in the description of "Subselect".

*include-columns*
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the UPDATE statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended at the end of the list of columns that are specified for *table-name* or *view-name*.

**INCLUDE**
Specifies a list of columns to be included in the intermediate result table of the UPDATE statement.

*column-name*
Specifies a column of the intermediate result table of the UPDATE statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name* (SQLSTATE 42711).

*data-type*
Specifies the data type of the include column. The data type must be one that is supported by the CREATE TABLE statement.

**SET**
Introduces the assignment of values to column names.

*assignment-clause*

*column-name*
Identifies a column to be updated. If extended indicator variables are not enabled, the *column-name* must identify an updatable column of the specified table, view, or nickname, or identify an INCLUDE column. The object ID column of a typed table is not updatable (SQLSTATE 428DZ). A column must not be specified more than once, unless it is followed by *..attribute-name* (SQLSTATE 42701).

If it specifies an INCLUDE column, the column name cannot be qualified.

For a Positioned UPDATE:

- If the *update-clause* was specified in the *select-statement* of the cursor, each column name in the *assignment-clause* must also appear in the *update-clause*.
- If the *update-clause* was not specified in the *select-statement* of the cursor and LANGLEVEL MIA or SQL92E was specified when the application was precompiled, the name of any updatable column may be specified.
- If the *update-clause* was not specified in the *select-statement* of the cursor and LANGLEVEL SAA1 was specified either explicitly or by default when the application was precompiled, no columns may be updated.

*..attribute-name*
Specifies the attribute of a structured type that is set (referred to as an *attribute assignment*. The *column-name* specified must be defined with a user-defined structured type (SQLSTATE 428DP). The attribute-name must

be an attribute of the structured type of *column-name* (SQLSTATE 42703). An assignment that does not involve the *..attribute-name* clause is referred to as a *conventional assignment*.

*expression*
Indicates the new value of the column. The expression is any expression of the type described in "Expressions". The expression cannot include an aggregate function except when it occurs within a scalar fullselect (SQLSTATE 42903).

An *expression* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated.

An expression cannot contain references to an INCLUDE column. If *expression* is a single host variable, the host variable can include an indicator variable that is enabled for extended indicator variables. If extended indicator variables are enabled, the extended indicator variable values of default (-5) or unassigned (-7) must not be used (SQLSTATE 22539) if either of the following statements is true:

- The expression is more complex than a single host variable with explicit casts
- The target column has data type of structured type

**NULL**
Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502). NULL cannot be the value in an attribute assignment (SQLSTATE 429B9) unless it is specifically cast to the data type of the attribute.

**DEFAULT**
Specifies that the default value should be used based on how the corresponding column is defined in the table. The value that is inserted depends on how the column was defined.

- If the column was defined as a generated column based on an expression, the column value will be generated by the system, based on the expression.
- If the column was defined using the IDENTITY clause, the value is generated by the database manager.
- If the column was defined using the WITH DEFAULT clause, the value is set to the default defined for the column (see *default-clause* in "ALTER TABLE").
- If the column was defined using the NOT NULL clause and the GENERATED clause was not used, or the WITH DEFAULT clause was not used, or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).
- If the column was defined using the ROW CHANGE TIMESTAMP clause, the value is generated by the database manager.

The only value that a generated column defined with the GENERATED ALWAYS clause can be set to is DEFAULT (SQLSTATE 428C9).

The DEFAULT keyword cannot be used as the value in an attribute assignment (SQLSTATE 429B9).

The DEFAULT keyword cannot be used as the value in an assignment for update on a nickname where the data source does not support DEFAULT syntax.

*row-fullselect*
A fullselect that returns a single row with the number of columns corresponding to the number of *column-name*s specified for assignment. The values are assigned to each corresponding *column-name*. If the result of the *row-fullselect* is no rows, then null values are assigned.

A *row-fullselect* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated. An error is returned if there is more than one row in the result (SQLSTATE 21000).

**WHERE**
Introduces a condition that indicates what rows are updated. You can omit the clause, give a search condition, or name a cursor. If the clause is omitted, all rows of the table, view or nickname are updated.

*search-condition*
Each *column-name* in the search condition, other than in a subquery, must name a column of the table, view or nickname. When the search condition includes a subquery in which the same table is the base object of both the UPDATE and the subquery, the subquery is completely evaluated before any rows are updated.

The search-condition is applied to each row of the table, view or nickname and the updated rows are those for which the result of the search-condition is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is executed only once, whereas a subquery with a correlated reference may have to be executed once for each row.

**CURRENT OF** *cursor-name*
Identifies the cursor to be used in the update operation. The *cursor-name* must identify a declared cursor, explained in "DECLARE CURSOR". The DECLARE CURSOR statement must precede the UPDATE statement in the program.

The specified table, view, or nickname must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR".)

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

This form of UPDATE cannot be used (SQLSTATE 42828) if the cursor references:

- A view on which an INSTEAD OF UPDATE trigger is defined
- A view that includes an OLAP function in the select list of the fullselect that defines the view
- A view that is defined, either directly or indirectly, using the WITH ROW MOVEMENT clause

**WITH**
Specifies the isolation level at which the UPDATE statement is executed.

**RR** Repeatable Read

**RS** Read Stability

**CS** Cursor Stability

**UR** Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. The WITH clause has no effect on nicknames, which always use the default isolation level of the statement.

## Rules

- ***Triggers:*** UPDATE statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the update values. If an update operation on a view causes an INSTEAD OF trigger to fire, validity, referential integrity, and constraints will be checked against the updates that are performed in the trigger, and not against the view that caused the trigger to fire, or its underlying tables.
- ***Assignment***: Update values are assigned to columns according to specific assignment rules.
- ***Validity***: The updated row must conform to any constraints imposed on the table (or on the base table of the view) by any unique index on an updated column.

  If a view is used that is not defined using WITH CHECK OPTION, rows can be changed so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

  If a view is used that is defined using WITH CHECK OPTION, an updated row must conform to the definition of the view. For an explanation of the rules governing this situation, see "CREATE VIEW".
- ***Check constraint***: Update value must satisfy the check-conditions of the check constraints defined on the table.

  An UPDATE to a table with check constraints defined has the constraint conditions for each column updated evaluated once for each row that is updated. When processing an UPDATE statement, only the check constraints referring to the updated columns are checked.
- ***Referential integrity***: The value of the parent unique keys cannot be changed if the update rule is RESTRICT and there are one or more dependent rows. However, if the update rule is NO ACTION, parent unique keys can be updated as long as every child has a parent key by the time the update statement completes. A non-null update value of a foreign key must be equal to a value of the primary key of the parent table of the relationship.
- ***XML values:*** When an XML column value is updated, the new value must be a well-formed XML document (SQLSTATE 2200M).
- ***Security policy:*** If the identified table or the base table of the identified view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow:
  - Write access to all protected columns that are being updated (SQLSTATE 42512)
  - Write access for any explicit value provided for a DB2SECURITYLABEL column for security policies that were created with the RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL option (SQLSTATE 23523)

– Read and write access to all rows that are being updated (SQLSTATE 42519)

The session authorization ID must also have been granted a security label for write access for the security policy if an implicit value is used for a DB2SECURITYLABEL column (SQLSTATE 23523), which can happen when:

– The DB2SECURITYLABEL column is not included in the list of columns that are to be updated (and so it will be implicitly updated to the security label for write access of the session authorization ID)

– A value for the DB2SECURITYLABEL column is explicitly provided but the session authorization ID does not have write access for that value, and the security policy is created with the OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option

- **Extended indicator variable usage**: If enabled, indicator variable values other than 0 (zero) through -7 must not be input (SQLSTATE 22010). Also, if enabled, the default and unassigned extended indicator variable values must not appear in contexts in which they are not supported (SQLSTATE 22539).
- **Extended indicator variables**: In the *assignment-clause* of an UPDATE statement, an *expression* that is a reference to a single host variable, or a host variable being explicitly cast can result in assigning an extended indicator variable value. Assigning an extended indicator variable-based value of unassigned has the effect of leaving the target column set to its current value, as if it had not been specified in the statement. Assigning an extended indicator variable-based value of default assigns the default value of the column. For information about default values of data types, see the description of the DEFAULT clause in the "CREATE TABLE" statement.

  If a target column is not updatable (for example, a column in a view that is defined as an expression), then it must be assigned the extended indicator variable-based value of unassigned (SQLSTATE 42808).

  If the target column is a column defined as GENERATED ALWAYS, then it must be assigned the DEFAULT keyword, or the extended indicator variable-based values of default or unassigned (SQLSTATE 428C9).

  The UPDATE statement must not assign all target columns to an extended indicator variable-based value of unassigned (SQLSTATE 22540).

## Examples

- *Example 1:* Change the job (JOB) of employee number (EMPNO) '000290' in the EMPLOYEE table to 'LABORER'.

```
UPDATE EMPLOYEE
   SET JOB = 'LABORER'
   WHERE EMPNO = '000290'
```

- *Example 2:* Increase the project staffing (PRSTAFF) by 1.5 for all projects that department (DEPTNO) 'D21' is responsible for in the PROJECT table.

```
UPDATE PROJECT
   SET PRSTAFF = PRSTAFF + 1.5
   WHERE DEPTNO = 'D21'
```

- *Example 3:* All the employees except the manager of department (WORKDEPT) 'E21' have been temporarily reassigned. Indicate this by changing their job (JOB) to the null value and their pay (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.

```
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

This statement could also be written as follows.

```
UPDATE EMPLOYEE
  SET (JOB, SALARY, BONUS, COMM) = (NULL, 0, 0, 0)
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

- *Example 4:* Update the salary and the commission column of the employee with employee number 000120 to the average of the salary and of the commission of the employees of the updated row's department, respectively.

```
UPDATE (SELECT EMPNO, SALARY, COMM,
  AVG(SALARY) OVER (PARTITION BY WORKDEPT),
  AVG(COMM) OVER (PARTITION BY WORKDEPT)
  FROM EMPLOYEE E) AS E(EMPNO, SALARY, COMM, AVGSAL, AVGCOMM)
SET (SALARY, COMM) = (AVGSAL, AVGCOMM)
WHERE EMPNO = '000120'
```

  The previous statement is semantically equivalent to the following statement, but requires only one access to the EMPLOYEE table, whereas the following statement specifies the EMPLOYEE table twice.

```
UPDATE EMPLOYEE EU
  SET (EU.SALARY, EU.COMM)
  =
(SELECT AVG(ES.SALARY), AVG(ES.COMM)
  FROM EMPLOYEE ES
  WHERE ES.WORKDEPT = EU.WORKDEPT)
  WHERE EU.EMPNO = '000120'
```

- *Example 5:* In a C program display the rows from the EMPLOYEE table and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in.

```
EXEC SQL  DECLARE C1 CURSOR FOR
               SELECT *
                 FROM EMPLOYEE
                 FOR UPDATE OF JOB;

EXEC SQL  OPEN C1;

EXEC SQL  FETCH C1 INTO ...     ;
if ( strcmp (change, "YES") == 0 )
  EXEC SQL  UPDATE EMPLOYEE
              SET JOB = :newjob
              WHERE CURRENT OF C1;

EXEC SQL  CLOSE C1;
```

- *Example 6:* These examples mutate attributes of column objects.

  Assume that the following types and tables exist:

```
CREATE TYPE POINT AS (X INTEGER, Y INTEGER)
  NOT FINAL WITHOUT COMPARISONS
  MODE DB2SQL

CREATE TYPE CIRCLE AS (RADIUS INTEGER, CENTER POINT)
  NOT FINAL WITHOUT COMPARISONS
  MODE DB2SQL

CREATE TABLE CIRCLES (ID INTEGER, OWNER VARCHAR(50), C CIRCLE
```

  The following example updates the CIRCLES table by changing the OWNER column and the RADIUS attribute of the CIRCLE column where the ID is 999:

```
UPDATE CIRCLES
  SET OWNER = 'Bruce'
    C..RADIUS = 5
  WHERE ID = 999
```

  The following example transposes the X and Y coordinates of the center of the circle identified by 999:

```
UPDATE CIRCLES
  SET C..CENTER..X = C..CENTER..Y,
    C..CENTER..Y = C..CENTER..X
  WHERE ID = 999
```

The following example is another way of writing both of the previous statements. This example combines the effects of both of the previous examples:

```
UPDATE CIRCLES
  SET (OWNER,C..RADIUS,C..CENTER..X,C..CENTER..Y) =
    ('Bruce',5,C..CENTER..Y,C..CENTER..X)
  WHERE ID = 999
```

- *Example 7:* Update the XMLDOC column of the DOCUMENTS table with DOCID '001' to the character string that is selected and parsed from the XMLTEXT table.

```
UPDATE DOCUMENTS SET XMLDOC =
  (SELECT XMLPARSE(DOCUMENT C1 STRIP WHITESPACE)
    FROM XMLTEXT WHERE TEXTID = '001')
WHERE DOCID = '001'
```

# Chapter 32. DELETE statement

The DELETE statement deletes rows from a table, nickname, or view, or the underlying tables, nicknames, or views of the specified fullselect.

Deleting a row from a nickname deletes the row from the data source object to which the nickname refers. Deleting a row from a view deletes the row from the table on which the view is based if no INSTEAD OF trigger is defined for the delete operation on this view. If such a trigger is defined, the trigger will be executed instead.

There are two forms of this statement:

- The *Searched* DELETE form is used to delete one or more rows (optionally determined by a search condition).
- The *Positioned* DELETE form is used to delete exactly one row (as determined by the current position of a cursor).

## Invocation

A DELETE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

## Authorization

To execute either form of this statement, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

- DELETE privilege on the table, view, or nickname from which rows are to be deleted
- CONTROL privilege on the table, view, or nickname from which rows are to be deleted
- DATAACCESS authority

To execute a Searched DELETE statement, the privileges held by the authorization ID of the statement must also include at least one of the following authorities for each table, view, or nickname referenced by a subquery:

- SELECT privilege
- CONTROL privilege
- DATAACCESS authority

If the package used to process the statement is precompiled with SQL92 rules (option LANGLEVEL with a value of SQL92E or MIA), and the searched form of a DELETE statement includes a reference to a column of the table or view in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

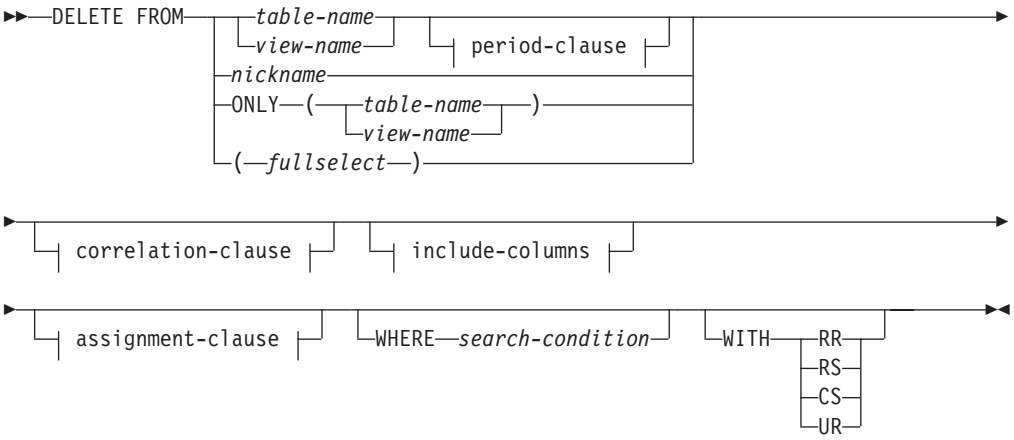
- SELECT privilege
- CONTROL privilege
- DATAACCESS authority

If the specified table or view is preceded by the ONLY keyword, the privileges held by the authorization ID of the statement must also include the SELECT privilege for every subtable or subview of the specified table or view.

Group privileges are not checked for static DELETE statements.

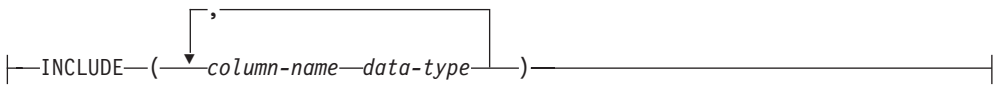
If the target of the delete operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

## Syntax (searched-delete)

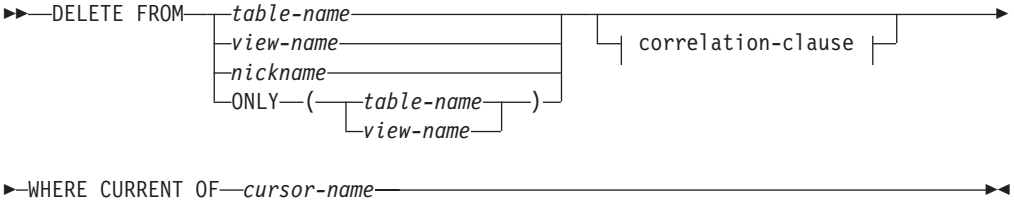



**period-clause:**

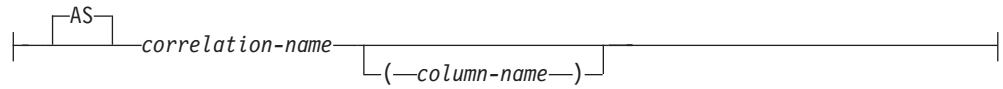

**include-columns:**




## Syntax (positioned-delete)



**correlation-clause:**




## Description

**FROM** *table-name*, *view-name*, *nickname*, **or** *(fullselect)*
Identifies the object of the delete operation. The name must identify one of the following objects:

- A table or view that exists in the catalog at the current server
- A table or view at a remote server specified using a remote-object-name

The object must not be a catalog table, a catalog view, a system-maintained materialized query table, or a read-only view.

If *table-name* is a typed table, rows of the table or any of its proper subtables may get deleted by the statement.

If *view-name* is a typed view, rows of the underlying table or underlying tables of the view's proper subviews may get deleted by the statement. If *view-name* is a regular view with an underlying table that is a typed table, rows of the typed table or any of its proper subtables may get deleted by the statement.

If the object of the delete operation is a fullselect, the fullselect must be deletable, as defined in the “Deletable views” Notes item in the description of the CREATE VIEW statement.

For additional restrictions related to temporal tables and use of a view or fullselect as the target of the delete operation, see "Considerations for a system-period temporal table" and "Considerations for an application-period temporal table" in the Notes section.

Only the columns of the specified table can be referenced in the WHERE clause. For a positioned DELETE, the associated cursor must also have specified the table or view in the FROM clause without using ONLY.

**FROM ONLY (***table-name***)**
Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

**FROM ONLY (***view-name***)**
Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

**period-clause**
Specifies that a period clause applies to the target of the delete operation.

If the target of the delete operation is a view, the following conditions apply to the view:

- The FROM clause of the outer fullselect of the view definition must include a reference, directly or indirectly, to an application-period temporal table (SQLSTATE 42724M).
- An INSTEAD OF DELETE trigger must not be defined for the view (SQLSTATE 428HY).

**FOR PORTION OF BUSINESS_TIME**
Specifies that the delete only applies to row values for the portion of the period in the row that is specified by the period clause. The BUSINESS_TIME period must exist in the table (SQLSTATE 4274M). FOR PORTION OF BUSINESS_TIME must not be specified if the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not NULL when the BUSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

**FROM** *value1* **TO** *value2*
Specifies that the delete applies to rows for the period specified from *value1* up to *value2*. No rows are deleted if *value1* is greater than or equal to *value2*, or if *value1* or *value2* is the null value (SQLSTATE 02000).

For the period specified with FROM *value1* TO *value2*, the BUSINESS_TIME period in a row in the target of the delete is in any of the following states:

- **Overlaps the beginning** of the specified period if the value of the begin column is less than *value1* and the value of the end column is greater than *value1*.
- **Overlaps the end**of the specified period if the value of the end column is greater than or equal to *value2* and the value of the begin column is less than *value2*.
- Is **fully contained** within the specified period if the value for the begin column for BUSINESS_TIME is greater than or equal to *value1* and the value for the corresponding end column is less than or equal to *value2*.
- Is **partially contained** in the specified period if the row overlaps the beginning of the specified period or the end of the specified period, but not both.
- **Fully overlaps** the specified period if the period in the row overlaps the beginning and end of the specified period.
- Is **not contained** in the period if both columns of BUSINESS_TIME are less than or equal to *value1* or greater than or equal to *value2*.

If the BUSINESS_TIME period in a row is not contained in the specified period, the row is not deleted. Otherwise, the delete is applied based on how the values in the columns of the BUSINESS_TIME period overlap the specified period as follows:

- If the BUSINESS_TIME period in a row is fully contained within the specified period, the row is deleted.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the beginning of the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the end of the specified period:

- The row is deleted.
- A row is inserted using the original values from the row, except that the begin column is set to *value2*.

• If the BUSINESS_TIME period in a row fully overlaps the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
  - An additional row is inserted using the original values from the row, except that the begin column is set to *value2*.

*value1* **and** *value2*
Each expression must return a value that has a date data type, timestamp data type, or a valid data type for a string representation of a date or timestamp (SQLSTATE 428HY). The result of each expression must be comparable to the data type of the columns of the specified period (SQLSTATE 42884). See the comparison rules described in "Assignments and comparisons".

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable
- Scalar function whose arguments are supported operands (though user-defined functions and non-deterministic functions cannot be used)
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

**correlation-clause**
Can be used within the *search-condition* to designate a table, view, nickname, or fullselect. For a description of *correlation-clause*, see "table-reference" in the description of "Subselect".

*include-columns*
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the DELETE statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended at the end of the list of columns that are specified for *table-name* or *view-name*.

**INCLUDE**
Specifies a list of columns to be included in the intermediate result table of the DELETE statement.

*column-name*
Specifies a column of the intermediate result table of the DELETE statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name* (SQLSTATE 42711).

*data-type*
Specifies the data type of the include column. The data type must be one that is supported by the CREATE TABLE statement.

*assignment-clause*
See the description of *assignment-clause* under the UPDATE statement. The same rules apply. The *include-columns* are the only columns that can be set using the *assignment-clause* (SQLSTATE 42703).

**WHERE**
Specifies a condition that selects the rows to be deleted. The clause can be omitted, a search condition specified, or a cursor named. If the clause is omitted, all rows of the table or view are deleted.

*search-condition*
Each *column-name* in the search condition, other than in a subquery must identify a column of the table or view.

The *search-condition* is applied to each row of the table, view, or nickname, and the deleted rows are those for which the result of the *search-condition* is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the *search condition* is applied to a row, and the results used in applying the *search condition*. In actuality, a subquery with no correlated references is executed once, whereas a subquery with a correlated reference may have to be executed once for each row. If a subquery refers to the object table of a DELETE statement or a dependent table with a delete rule of CASCADE or SET NULL, the subquery is completely evaluated before any rows are deleted.

**CURRENT OF** *cursor-name*
Identifies a cursor that is defined in a DECLARE CURSOR statement of the program. The DECLARE CURSOR statement must precede the DELETE statement.

The table, view, or nickname named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR".)

When the DELETE statement is executed, the cursor must be positioned on a row: that row is the one deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

**WITH**
Specifies the isolation level used when locating the rows to be deleted.

**RR** Repeatable Read

**RS** Read Stability

**CS** Cursor Stability

**UR** Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. The WITH clause has no effect on nicknames, which always use the default isolation level of the statement.

## Rules

- ***Triggers:*** DELETE statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the deleted rows. If a DELETE statement on a view causes an INSTEAD OF

trigger to fire, referential integrity will be checked against the updates performed in the trigger, and not against the underlying tables of the view that caused the trigger to fire.

- ***Referential integrity:*** If the identified table or the base table of the identified view is a parent, the rows selected for delete must not have any dependents in a relationship with a delete rule of RESTRICT, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT.

  If the delete operation is not prevented by a RESTRICT delete rule, the selected rows are deleted. Any rows that are dependents of the selected rows are also affected:

  – The nullable columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET NULL are set to the null value.
  – Any rows that are their dependents in a relationship with a delete rule of CASCADE are also deleted, and the preceding rules apply, in turn, to those rows.

  The delete rule of NO ACTION is checked to enforce that any non-null foreign key refers to an existing parent row after the other referential constraints have been enforced.
- ***Security policy:*** If the identified table or the base table of the identified view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow:
  – Write access to all protected columns (SQLSTATE 42512)
  – Read and write access to all of the rows that are selected for deletion (SQLSTATE 42519)

## Examples

- *Example 1:* Delete department (DEPTNO) 'D11' from the DEPARTMENT table.

```
DELETE FROM DEPARTMENT
  WHERE DEPTNO = 'D11'
```

- *Example 2:* Delete all the departments from the DEPARTMENT table (that is, empty the table).

```
DELETE FROM DEPARTMENT
```

- *Example 3:* Delete from the EMPLOYEE table any sales rep or field rep who didn't make a sale in 1995.

```
DELETE FROM EMPLOYEE
  WHERE LASTNAME NOT IN
    (SELECT SALES_PERSON
      FROM SALES
      WHERE YEAR(SALES_DATE)=1995)
      AND JOB IN ('SALESREP','FIELDREP')
```

- 
- *Example 4:* Delete all the duplicate employee rows from the EMPLOYEE table. An employee row is considered to be a duplicate if the last names match. Keep the employee row with the smallest first name in lexical order.

```
DELETE FROM
  (SELECT ROWNUMBER() OVER (PARTITION BY LASTNAME ORDER BY FIRSTNME)
    FROM EMPLOYEE) AS E(RN)
    WHERE RN > 1
```

# Chapter 33. SQL queries

A *query* specifies a result table. A query is a component of certain SQL statements.

The three forms of a query are:
- subselect
- fullselect
- select-statement.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:
- For each table or view identified in the query, one of the following authorities:
  - SELECT privilege on the table or view
  - CONTROL privilege on the table or view
- DATAACCESS authority

For each global variable used as an expression in the query, the privileges held by the authorization ID of the statement must include one of the following authorities:
- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

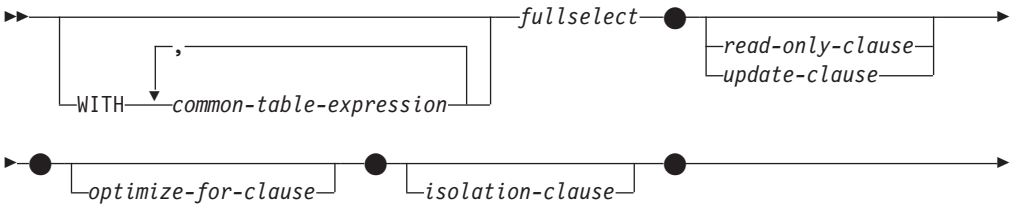
If the query contains an SQL data change statement, the authorization requirements of that statement also apply to the query.

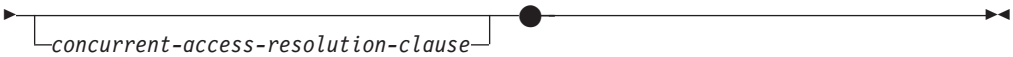
Group privileges, with the exception of PUBLIC, are not checked for queries that are contained in static SQL statements or DDL statements.

For nicknames, authorization requirements of the data source for the object referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

## select-statement

The *select-statement* is the form of a query that can be specified in a DECLARE CURSOR statement, either directly, or prepared and then referenced. It can also be issued through the use of dynamic SQL statements, causing a result table to be displayed on the user's screen. The table specified by a *select-statement* is the result of the fullselect.






The authorization for a *select-statement* is described in the Authorization section in "SQL queries".

## Examples of select-statement queries

The following examples illustrate the select-statement query.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

- *Example 2:* Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

```
SELECT PROJNAME, PRSTDATE, PRENDATE
 FROM PROJECT
 ORDER BY PRENDATE DESC
```

- *Example 3:* Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
   FROM EMPLOYEE
   GROUP BY WORKDEPT
   ORDER BY 2
```

- *Example 4:* Declare a cursor named UP_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT PROJNO, PRSTDATE, PRENDATE
              FROM PROJECT
              FOR UPDATE OF PRSTDATE, PRENDATE;
```

- *Example 5:* This example names the expression SAL+BONUS+COMM as TOTAL_PAY

```
SELECT SALARY+BONUS+COMM AS TOTAL_PAY
  FROM EMPLOYEE
  ORDER BY TOTAL_PAY
```

- *Example 6:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

  Using a common table expression for this case saves the processing resources of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the fullselect, only the rows for the department of the sales representatives are considered by the view.

```
WITH
   DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
       (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
         FROM EMPLOYEE OTHERS
         GROUP BY OTHERS.WORKDEPT
       ),
   DINFOMAX AS
       (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
```

```
      DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
WHERE THIS_EMP.JOB = 'SALESREP'
AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

- *Example 7:* Given two tables, EMPLOYEE and PROJECT, replace employee SALLY with a new employee GEORGE, assign all projects lead by SALLY to GEORGE, and return the names of the updated projects.

```
WITH
  NEWEMP AS (SELECT EMPNO FROM NEW TABLE
              (INSERT INTO EMPLOYEE(EMPNO, FIRSTNME)
                  VALUES(NEXT VALUE FOR EMPNO_SEQ, 'GEORGE'))),
  OLDEMP AS (SELECT EMPNO FROM EMPLOYEE WHERE FIRSTNME = 'SALLY'),
  UPPROJ AS (SELECT PROJNAME FROM NEW TABLE
              (UPDATE PROJECT
                  SET RESPEMP = (SELECT EMPNO FROM NEWEMP)
                  WHERE RESPEMP = (SELECT EMPNO FROM OLDEMP))),
  DELEMP AS (SELECT EMPNO FROM OLD TABLE
              (DELETE FROM EMPLOYEE
                  WHERE EMPNO = (SELECT EMPNO FROM OLDEMP)))
SELECT PROJNAME FROM UPPROJ;
```

- *Example 8:* Retrieve data from the DEPT table. That data will later be updated with a searched update, and will be locked when the query executes.

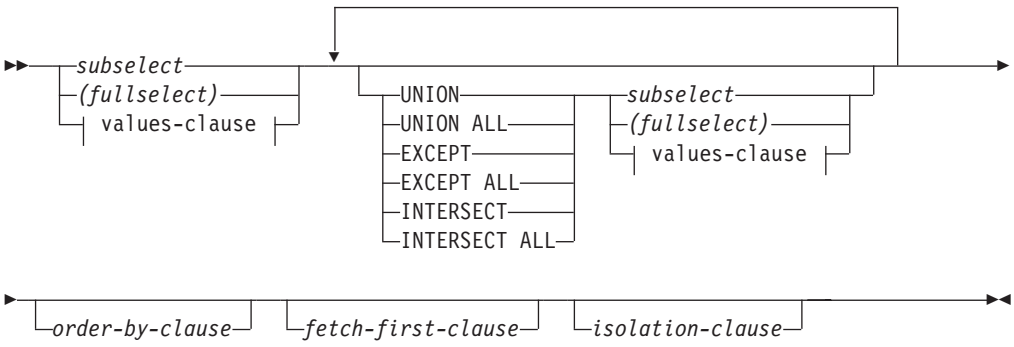
```
SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPT
  WHERE ADMRDEPT ='A00'
  FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS
```

- *Example 9:* Select all columns and rows from the EMPLOYEE table. If another transaction is concurrently updating, deleting, or inserting data in the EMPLOYEE table, the select operation will wait to get the data until after the other transaction is completed.

```
SELECT * FROM EMPLOYEE WAIT FOR OUTCOME
```

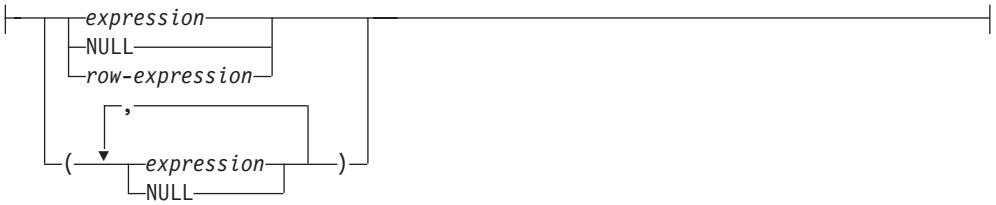
## fullselect

The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a statement. A fullselect that is a component of a predicate is called a *subquery*, and a fullselect that is enclosed in parentheses is sometimes called a subquery.



**values-clause:**


**values-row:**




The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection.

A fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect or values-clause.

The authorization for a *fullselect* is described in the Authorization section in "SQL queries".

*values-clause*
Derives a result table by specifying the actual values, using expressions or row expressions, for each column of a row in the result table. Multiple rows may be specified. If multiple rows are specified, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539). The result type of any expression in the *values-clause* cannot be a row type (SQLSTATE 428H2).

NULL can only be used with multiple specifications of *values-row*, either as the column value of a single column result table or within a *row-expression*, and at least one row in the same column must not be NULL (SQLSTATE 42608).

A *values-row* is specified by:

- A single expression for a single column result table
- *n* expressions (or NULL) separated by commas and enclosed in parentheses, where *n* is the number of columns in the result table or, a row expression for a multiple column result table.

A multiple row VALUES clause must have the same number of columns in each *values-row* (SQLSTATE 42826).

The following examples show *values-clause* and their meaning.

```
VALUES (1),(2),(3)          - 3 rows of 1 column
VALUES 1, 2, 3              - 3 rows of 1 column
VALUES (1, 2, 3)            - 1 row of 3 columns
VALUES (1,21),(2,22),(3,23) - 3 rows of 2 columns
```

A *values-clause* that is composed of *n* specifications of *values-row*, $RE_1$ to $RE_n$, where *n* is greater than 1, is equivalent to:

$RE_1$ UNION ALL $RE_2$ ... UNION ALL $RE_n$

This means that the corresponding columns of each *values-row* must be comparable (SQLSTATE 42825).

**UNION or UNION ALL**
Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

**EXCEPT or EXCEPT ALL**
Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT.

**INTERSECT or INTERSECT ALL**
Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

*order-by-clause*
See "subselect" for details of the *order-by-clause*. A fullselect that contains an ORDER BY clause cannot be specified in (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

**Note:** An ORDER BY clause in a fullselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

*fetch-first-clause*
See "subselect" for details of the *fetch-first-clause*. A fullselect that contains a FETCH FIRST clause cannot be specified in (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

**Note:** A FETCH FIRST clause in a fullselect does not affect the number of rows returned by a query. A FETCH FIRST clause only affects the number of rows returned if it is specified in the outermost fullselect.

*isolation-clause*
See "subselect" for details of the *isolation-clause*. If *isolation-clause* is specified for a fullselect and it could apply equally to a subselect of the fullselect, *isolation-clause* is applied to the fullselect. For example, consider the following query.

```
SELECT NAME FROM PRODUCT
UNION
SELECT NAME FROM CATALOG
WITH UR
```

Even though the isolation clause WITH UR could apply only to the subselect SELECT NAME FROM CATALOG, it is applied to the whole fullselect.

The number of columns in the result tables R1 and R2 must be the same (SQLSTATE 42826). If the ALL keyword is not specified, R1 and R2 must not

include any columns having a data type of CLOB, DBCLOB, BLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

The column name of the *n*th column of the result table is the name of the *n*th column of R1 if it is named. Otherwise, the *n*th column of the result table is unnamed. If the fullselect is used as a select-statement, a generated name is provided when the statement is described. The generated name cannot be used in other parts of the SQL statement such as the ORDER BY clause or the UPDATE clause. The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

**Duplicate rows:** Two rows are duplicates if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal, and two decimal floating-point representations of the same number are considered equal. For example, 2.00 and 2.0 have the same value (2.00 and 2.0 compare as equal) but have different exponents, which allows you to represent both 2.00 and 2.0. So, for example, if the result table of a UNION operation contains a decimal floating-point column and multiple representations of the same number exist, the one that is returned (for example, 2.00 or 2.0) is unpredictable.

When multiple operations are combined in an expression, operations within parentheses are performed first. If there are no parentheses, the operations are performed from left to right with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.

| R1 | R2 | UNION ALL | UNION | EXCEPT ALL | EXCEPT | INTER-SECT ALL | INTER-SECT |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 |
| 1 | 1 | 1 | 2 | 2 | 5 | 1 | 3 |
| 1 | 3 | 1 | 3 | 2 | | 3 | 4 |
| 2 | 3 | 1 | 4 | 2 | | 4 | |
| 2 | 3 | 1 | 5 | 4 | | | |
| 2 | 3 | 2 | | 5 | | | |
| 3 | 4 | 2 | | | | | |
| 4 | | 2 | | | | | |
| 4 | | 3 | | | | | |
| 5 | | 3 | | | | | |
| | | 3 | | | | | |
| | | 3 | | | | | |
| | | 3 | | | | | |
| | | 4 | | | | | |
| | | 4 | | | | | |

| R1 | R2 | UNION ALL | UNION | EXCEPT ALL | EXCEPT | INTER-SECT ALL | INTER-SECT |
|---|---|---|---|---|---|---|---|
| | | 4 | | | | | |
| | | 5 | | | | | |

## Examples of fullselect queries

The following examples illustrate fullselect queries.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

- *Example 2:* List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E' **or** who are assigned to projects in the EMP_ACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```
SELECT EMPNO
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 3:* Make the same query as in example 2, and, in addition, "tag" the rows from the EMPLOYEE table with 'emp' and the rows from the EMP_ACT table with 'emp_act'. Unlike the result from example 2, this query might return the same EMPNO more than once, identifying which table it came from by the associated "tag".

```
SELECT EMPNO, 'emp'
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 4:* Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```
SELECT EMPNO
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 5:* Make the same query as in Example 3, only include an additional two employees currently not in any table and tag these rows as "new".

```
SELECT EMPNO, 'emp'
   FROM EMPLOYEE
   WHEREWORKDEPTLIKE 'E%'
UNION
SELECT EMPNO, 'emp_act'
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
UNION
   VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')
```

- *Example 6:* This example of EXCEPT produces all rows that are in T1 but not in T2.

```
(SELECT * FROM T1)
EXCEPT ALL
(SELECT * FROM T2)
```

If no null values are involved, this example returns the same results as

```
SELECT ALL *
   FROM T1
   WHERE NOT EXISTS (SELECT * FROM T2
                     WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

- *Example 7:* This example of INTERSECT produces all rows that are in both tables T1 and T2, removing duplicates.

```
(SELECT * FROM T1)
INTERSECT
(SELECT * FROM T2)
```

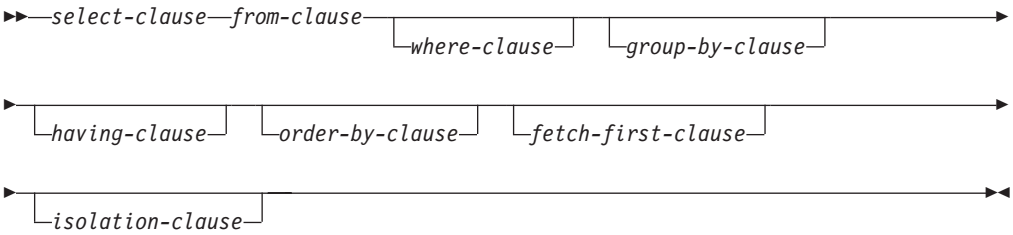
If no null values are involved, this example returns the same result as

```
SELECT DISTINCT * FROM T1
   WHERE EXISTS (SELECT * FROM T2
                 WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

where C1, C2, and so on represent the columns of T1 and T2.

## subselect

The *subselect* is a component of the fullselect.




A subselect specifies a result table derived from the tables, views or nicknames identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method used to perform the derivation can be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they might or might not be executed.)

The authorization for a *subselect* is described in the Authorization section in "SQL queries".

The clauses of the subselect are processed in the following sequence:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. FETCH FIRST clause

A subselect that contains an ORDER BY or FETCH FIRST clause cannot be specified:

- In the outermost fullselect of a view.
- In the outer fullselect of a materialized query table.
- Unless the subselect is enclosed in parenthesis.

For example, the following is not valid (SQLSTATE 428FJ):

```
SELECT * FROM T1
   ORDER BY C1
UNION
SELECT * FROM T2
   ORDER BY C1
```

The following example *is* valid:

```
(SELECT * FROM T1
   ORDER BY C1)
UNION
(SELECT * FROM T2
   ORDER BY C1)
```

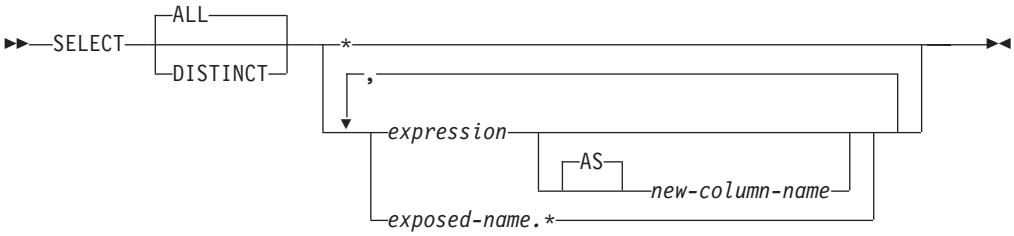
**Note:** An ORDER BY clause in a subselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

For details about the clauses in the subselect query, refer to the following topics:

- "select-clause"
- "from-clause" on page 289
- "where-clause" on page 309
- "group-by-clause" on page 309
- "having-clause" on page 315
- "order-by-clause" on page 316
- "fetch-first-clause" on page 319
- "isolation-clause (subselect query)" on page 319

## select-clause

The SELECT clause specifies the columns of the final result table.




The column values are produced by the application of the *select list* to the final result table, *R*. The select list is the names or expressions specified in the SELECT clause, and *R* is the result of the previous operation of the subselect. For example, if the only clauses specified are SELECT, FROM, and WHERE, *R* is the result of that WHERE clause.

**ALL**
Retains all rows of the final result table, and does not eliminate redundant duplicates. This is the default.

**DISTINCT**
Eliminates all but one of each set of duplicate rows of the final result table. If DISTINCT is used, no string column of the result table can be a LOB type, distinct type based on LOB, or structured type. DISTINCT can be used more than once in a subselect. This includes SELECT DISTINCT, the use of DISTINCT in an aggregate function of the select list or HAVING clause, and subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value in the second. For determining duplicates, two null values are considered equal, and two different decimal floating-point representations of the same number are considered equal. For example, -0 is equal to +0 and 2.0 is equal to 2.00. Each of the decimal floating-point special values are also considered equal: -NAN equals -NAN, -SNAN equals -SNAN, -INFINITY equals -INFINITY, INFINITY equals INFINITY, SNAN equals SNAN, and NAN equals NAN.

When the data type of a column is decimal floating-point, and multiple representations of the same number exist in the column, the particular value that is returned for a SELECT DISTINCT can be any one of the representations in the column.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

## Select list notation

* Represents a list of names that identify the columns of table *R*, excluding any columns defined as IMPLICITLY HIDDEN. The first name in the list identifies the first column of *R*, the second name identifies the second column of *R*, and so on.

The list of names is established when the program containing the SELECT clause is bound. Hence * (the asterisk) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

*expression*
Specifies the values of a result column. Can be any expression that is a valid SQL language element, but commonly includes column names. Each column name used in the select list must unambiguously identify a column of R. The result type of the expression cannot be a row type (SQLSTATE 428H2).

*new-column-name* **or AS** *new-column-name*
Names or renames the result column. The name must not be qualified and does not have to be unique. Subsequent usage of column-name is limited as follows:

- A *new-column-name* specified in the AS clause can be used in the order-by-clause, provided the name is unique.
- A *new-column-name* specified in the AS clause of the select list cannot be used in any other clause within the subselect (where-clause, group-by-clause or having-clause).
- A *new-column-name* specified in the AS clause cannot be used in the update-clause.

- A *new-column-name* specified in the AS clause is known outside the fullselect of nested table expressions, common table expressions and CREATE VIEW.

*exposed-name.**
Represents the list of names that identify the columns of the result table identified by *exposed-name*, excluding any columns defined as IMPLICITLY HIDDEN. The *exposed-name* can be a table name, view name, nickname, or correlation name, and must designate a table, view or nickname named in the FROM clause. The first name in the list identifies the first column of the table, view or nickname, the second name in the list identifies the second column of the table, view or nickname, and so on.

The list of names is established when the statement containing the SELECT clause is bound. Therefore, * does not identify any columns that have been added to a table after the statement has been bound.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared), and cannot exceed 500 for a 4K page size or 1012 for an 8K, 16K, or 32K page size.

## Limitations on string columns

For restrictions using varying-length character strings on the select list, see "Character strings" on page 120.

## Applying the select list

Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is used. The results are described in two separate lists.

## If GROUP BY or HAVING is used

- An expression *X* (not an aggregate function) used in the select list must have a GROUP BY clause with:
  - a *grouping-expression* in which each expression or column-name unambiguously identifies a column of R (see "group-by-clause" on page 309) or
  - each column of R referenced in *X* as a separate *grouping-expression*.
- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the aggregate functions in the select list.

## If neither GROUP BY nor HAVING is used

- Either the select list must not include any aggregate functions, or each *column-name* in the select list must be specified within an aggregate function or must be a correlated column reference.
- If the select does not include aggregate functions, then the select list is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list is a list of aggregate functions, then R is the source of the arguments of the functions and the result of applying the select list is one row.

In either case the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

## Null attributes of result columns

Result columns do not allow null values if they are derived from:

- A column that does not allow null values
- A constant
- The COUNT or COUNT_BIG function
- A host variable that does not have an indicator variable
- A scalar function or expression that does not include an operand where nulls are allowed

Result columns allow null values if they are derived from:

- Any aggregate function except COUNT or COUNT_BIG
- A column where null values are allowed
- A scalar function or expression that includes an operand where nulls are allowed
- A NULLIF function with arguments containing equal values
- A host variable that has an indicator variable, an SQL parameter, an SQL variable, or a global variable
- A result of a set operation if at least one of the corresponding items in the select list is nullable
- An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with **dft_sqlmathwarn** set to Yes
- A scalar subselect
- A dereference operation
- A GROUPING SETS *grouping-expression*

## Names of result columns

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), then the result column name is the unqualified name of that column.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single SQL variable or SQL parameter (without any functions or operators), then the result column name is the unqualified name of that SQL variable or SQL parameter.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived using a dereference operation, then the result column name is the unqualified name of the target column of the dereference operation.
- All other result column names are unnamed. The system assigns temporary numbers (as character strings) to these columns.

## Data types of result columns

Each column of the result of SELECT acquires a data type from the expression from which it is derived.

| When the expression is ... | The data type of the result column is ... |
|---|---|
| the name of any numeric column | the same as the data type of the column, with the same precision and scale for DECIMAL columns, or the same precision for DECFLOAT columns. |
| a constant | the same as the data type of the constant. |
| the name of any numeric variable | the same as the data type of the variable, with the same precision and scale for DECIMAL variables, or the same precision for DECFLOAT variables. |
| the name of any string column | the same as the data type of the column, with the same length attribute. |
| the name of any string variable | the same as the data type of the variable, with the same length attribute; if the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string. |
| the name of a datetime column | the same as the data type of the column. |
| the name of a user-defined type column | the same as the data type of the column. |
| the name of a reference type column | the same as the data type of the column. |

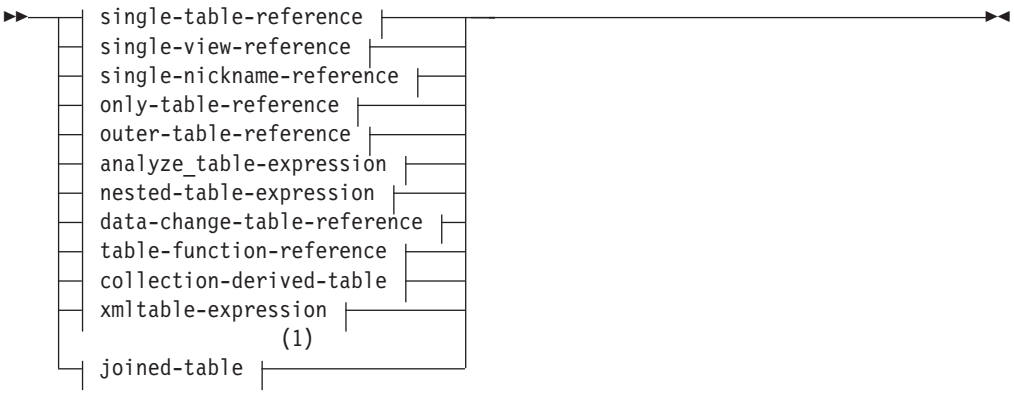
## from-clause

The FROM clause specifies an intermediate result table.

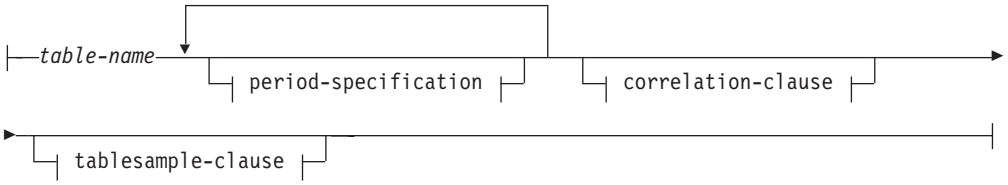

If only one *table-reference* is specified, the intermediate result table is the result of that *table-reference*. If more than one *table-reference* is specified, the intermediate result table consists of all possible combinations of the rows of the specified *table-reference* (the Cartesian product). Each row of the result is a row from the first *table-reference* concatenated with a row from the second *table-reference*, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table references. For a description of *table-reference*, see "table-reference."

### table-reference

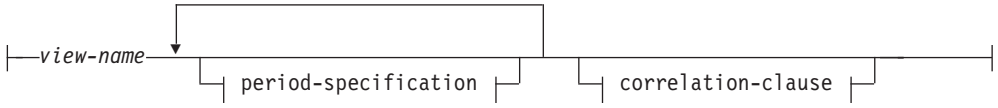
A *table-reference* specifies an intermediate result table.



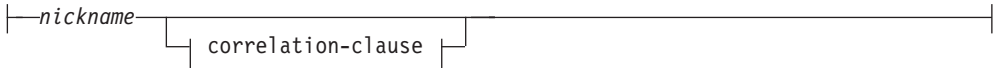

**single-table-reference:**




**single-view-reference:**



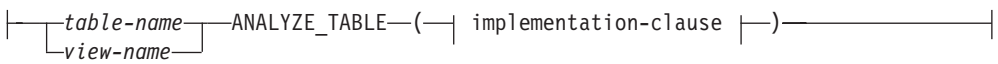

**single-nickname-reference:**




**only-table-reference:**

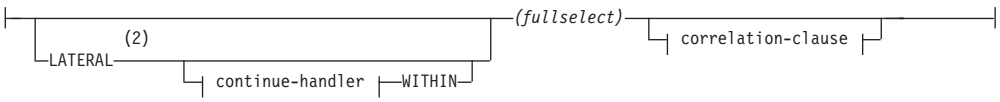

**outer-table-reference:**


**analyze_table-expression:**



**implementation-clause:**


**nested-table-expression:**




**data-change-table-reference:**

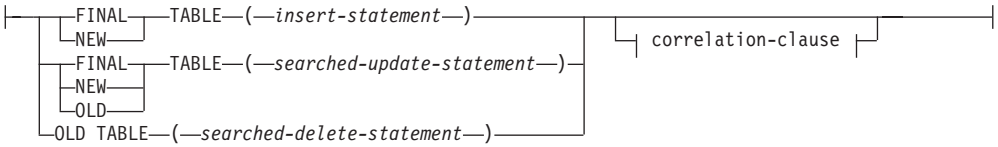
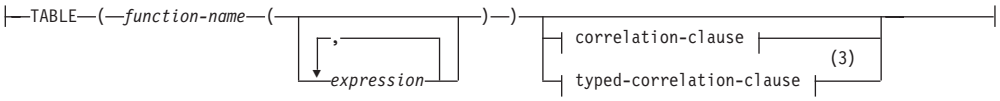
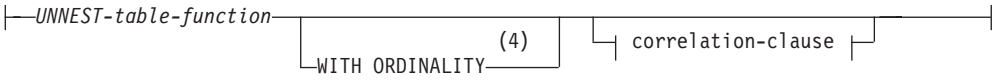


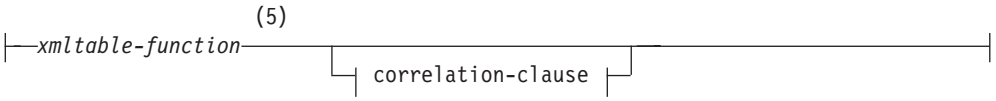

**table-function-reference:**



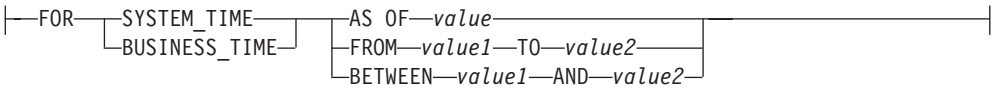

**collection-derived-table:**



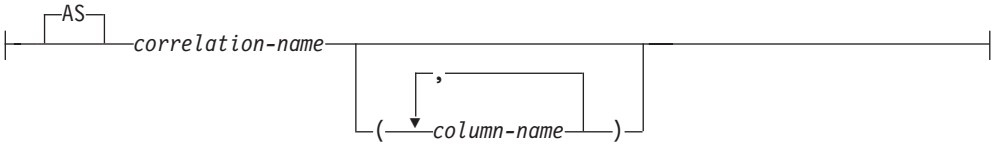

**xmltable-expression:**



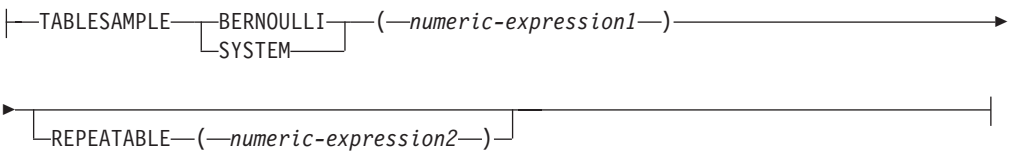

**period-specification:**



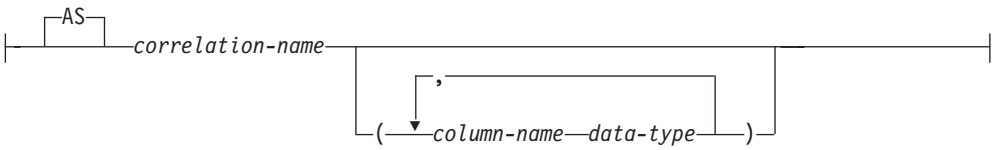

**correlation-clause:**



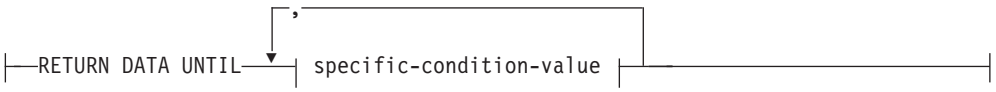
**tablesample-clause:**



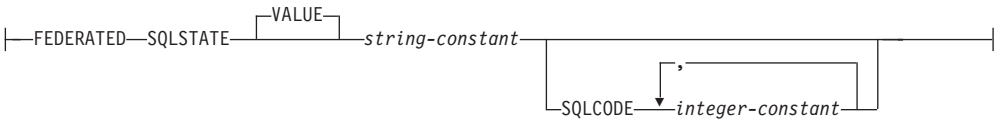

**typed-correlation-clause:**




**continue-handler:**




**specific-condition-value:**




**Notes:**

1 The syntax for joined-table is covered in a separate topic; refer to "joined-table" on page 307.

2 TABLE can be specified in place of LATERAL.

3 The typed-correlation-clause is required for generic table functions. This clause cannot be specified for any other table functions.

4 WITH ORDINALITY can be specified only if the argument to the UNNEST table function is one or more ordinary array variables or functions with ordinary array return types; an associative array variable or function with an associative array return type cannot be specified (SQLSTATE 428HT).

5 An XMLTABLE function can be part of a table-reference. In this case, subexpressions within the XMLTABLE expression are in-scope of prior range variables in the FROM clause. For more information, see the description of "XMLTABLE".

A *table-reference* specifies an intermediate result table.

- If a single-table-reference is specified without a period-specification or a tablesample-clause, the intermediate result table is the rows of the table. If a period-specification is specified, the intermediate result table consists of the rows of the temporal table where the period matches the specification. If a tablesample-clause is specified, the intermediate result table consists of a sampled subset of the rows of the table.

- If a single-view-reference is specified without a period-specification, the intermediate result table is that view. If a period-specification is specified, temporal table references in the view consider only the rows where the period matches the specification.
- If a single-nickname-reference is specified, the intermediate result table is the data from the data source for that nickname.
- If an only-table-reference is specified, the intermediate result table consists of only the rows of the specified table or view without considering the applicable subtables or subviews.
- If an outer-table-reference is specified, the intermediate result table represents a virtual table based on all the subtables of a typed table or the subviews of a typed view.
- If an analyze_table-expression is specified, the result table contains the result of executing a specific data mining model by using an in-database analytics provider, a named model implementation, and input data.
- If a nested-table-expression is specified, the result table is the result of the specified fullselect.
- If a data-change-table-reference is specified, the intermediate result table is the set of rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause.
- If a table-function-reference is specified, the intermediate result table is the set of rows that are returned by the table function.
- If a collection-derived-table is specified, the intermediate result table is the set of rows that are returned by the UNNEST function.
- If an xmltable-expression is specified, the intermediate result table is the set of rows that are returned by the XMLTABLE function.
- If a joined-table is specified, the intermediate result table is the result of one or more join operations. For more information, see "joined-table" on page 307.

*single-table-reference*

Each *table-name* specified as a table-reference must identify an existing table at the application server or an existing table at a remote server specified using a remote-object-name. The intermediate result table is the result of the table. If the *table-name* references a typed table, the intermediate result table is the UNION ALL of the table with all its subtables, with only the columns of the *table-name*. A period-specification can be used with a temporal table to specify the period from which the rows are returned as the intermediate result table. A tablesample-clause can be used to specify that a sample of the rows be returned as the intermediate result table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value *CTST* and *table-name* identifies a system-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
table-name FOR SYSTEM_TIME AS OF CTST
```

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value *CTBT* and *table-name* identifies an application-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
table-name FOR BUSINESS_TIME AS OF CTBT
```

*single-view-reference*

Each *view-name* specified as a table-reference must identify one of the following objects:

- An existing view at the application server
- A view at a remote server specified using a remote-object-name
- The *table-name* of a common table expression

The intermediate result table is the result of the view or common table expression. If the *view-name* references a typed view, the intermediate result table is the UNION ALL of the view with all its subviews, with only the columns of the *view-name*. A period-specification can be used with a view defined over a temporal table to specify the period from which the rows are returned as the intermediate result table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value *CTST*, and *view-name* identifies a system-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
view-name FOR SYSTEM_TIME AS OF CTST
```

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value *CTBT*, and *view-name* identifies an application-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
view-name FOR BUSINESS_TIME AS OF CTBT
```

*single-nickname-reference*

Each *nickname* specified as a table-reference must identify an existing nickname at the application server. The intermediate result table is the result of the nickname.

*only-table-reference*

The use of ONLY(*table-name*) or ONLY(*view-name*) means that the rows of the applicable subtables or subviews are not included in the intermediate result table. If the *table-name* used with ONLY does not have subtables, then ONLY(*table-name*) is equivalent to specifying *table-name*. If the *view-name* used with ONLY does not have subviews, then ONLY(*view-name*) is equivalent to specifying *view-name*.

The use of ONLY requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

*outer-table-reference*

The use of OUTER(*table-name*) or OUTER(*view-name*) represents a virtual table. If the *table-name* or *view-name* used with OUTER does not have subtables or subviews, then specifying OUTER is equivalent to not specifying OUTER. If the *table-name* does have subtables, the intermediate result table from OUTER(*table-name*) is derived from *table-name* as follows:

- The columns include the columns of *table-name* followed by the additional columns introduced by each of its subtables, if any. The additional columns are added on the right, traversing the subtable hierarchy in depth-first order. Subtables that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *table-name* and all the rows of its subtables. Null values are returned for columns that are not in the subtable for the row.

If the *view-name* does have subviews, the intermediate result table from OUTER(*view-name*) is derived from *view-name* as follows:

- The columns include the columns of *view-name* followed by the additional columns introduced by each of its subviews, if any. The additional columns are added on the right, traversing the subview hierarchy in depth-first order. Subviews that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *view-name* and all the rows of its subviews. Null values are returned for columns that are not in the subview for the row.

The use of OUTER requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

*analyze_table-expression*

*table-name* | *view-name*
The *table-name* or *view-name* variable must identify an existing table or view or identify the *table-name* of a common table expression that you define preceding the fullselect containing the table-reference. You can specify a nickname. However, in-database analytics are intended for local data, and retrieving the data for a nickname from another data source does not take advantage of the intended performance benefits.

**ANALYZE_TABLE**
Returns the result of executing a specific data mining model by using an in-database analytics provider, a named model implementation, and input data. A query referencing the ANALYZE_TABLE parameter cannot be a static SQL statement or a data definition language (DDL) statement. Input or output values cannot be of type CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BLOB, CLOB, DBCLOB, NCLOB, XML, or DB2SECURITYLABEL.

**IMPLEMENTATION '***string***'**
Specifies how the expression is to be evaluated. The *string* parameter is a string constant whose maximum length is 1024 bytes. The specified value is used to establish a session with an in-database analytic provider. When you specify SAS as the provider, you must specify values for the following case-insensitive parameters:

**PROVIDER**
Currently, the only supported provider value is SAS.

**ROUTINE_SOURCE_TABLE**
Specifies a user table containing the DS2 code (and, optionally, any required format or metadata) to implement the algorithm that is specified by the ROUTINE_SOURCE_NAME parameter. DS2 is a procedural language processor for SAS, designed for data modeling, stored procedures, and data extraction, transformation, and load (ETL) processing.

The routine source table has a defined structure (see the examples at the end of the "*analyze_table-expression*" section) and, in a partitioned database environment, must be on the catalog database partition. The table cannot be a global temporary table. The MODELDS2 column for a particular row must not be empty or contain the null value. If the value of the MODELFORMATS or MODELMETADATA column is not null, the value must have a length greater than 0. If you do not specify a table schema name, the value of the CURRENT SCHEMA special register is used.

**ROUTINE_SOURCE_NAME**
Specifies the name of the algorithm to use.

For example:

```
IMPLEMENTATION
  'PROVIDER=SAS;
  ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
  ROUTINE_SOURCE_NAME=SCORING_FUN1;'
```

If the table name, schema name, or algorithm name contains lowercase or mixed-case letters, specify delimited identifiers, as shown in the following example:

```
IMPLEMENTATION
  'PROVIDER=SAS;
  ROUTINE_SOURCE_TABLE="ETLin"."Source_Table";
  ROUTINE_SOURCE_NAME="Scoring_Fun1";'
```

The following examples show you how to use the ANALYZE_TABLE expression.

SAS tooling helps you to define a table to store model implementations for scoring functions. A row in this table stores an algorithm that is written in DS2, with any required SAS format information and metadata. The MODELNAME column serves as the primary key. For a particular value of the ROUTINE_SOURCE_NAME parameter, at most one row is retrieved from the table that the ROUTINE_SOURCE_TABLE parameter specifies. For example:

```
CREATE TABLE ETLIN.SOURCE_TABLE (
  MODELNAME VARCHAR(128) NOT NULL PRIMARY KEY,
  MODELDS2 BLOB(4M) NOT NULL,
  MODELFORMATS BLOB(4M),
  MODELMETADATA BLOB(4M)
);
```

The MODELNAME column contains the name of the algorithm. The MODELDS2 column contains the DS2 source code that implements the algorithm. The MODELFORMATS column contains the aggregated SAS format definition that the algorithm requires. If the algorithm does not require a SAS format, this column contains the null value. The MODELMETADATA column contains any additional metadata that the algorithm requires. If the algorithm does not require any additional metadata, this column contains the null value. If the SAS EP installer creates the table, it might include additional columns.

- Use the data in columns C1 and C2 in table T1 as input data with the scoring model SCORING_FUN1, whose implementation is stored in ETLIN.SOURCE_TABLE:

```
WITH sas_score_in (c1,c2) AS
  (SELECT c1,c2 FROM t1)
  SELECT *
    FROM sas_score_in ANALYZE_TABLE(
      IMPLEMENTATION
        'PROVIDER=SAS;
         ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
         ROUTINE_SOURCE_NAME=SCORING_FUN1;');
```

- Use all the data in the table T2 with the scoring model SCORING_FUN2, whose implementation is stored in ETLIN.SOURCE_TABLE:

```
SELECT *
  FROM t2 ANALYZE_TABLE(
    IMPLEMENTATION
      'PROVIDER=SAS;
       ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
       ROUTINE_SOURCE_NAME=SCORING_FUN2;');
```

- Use all the data in view V1 with the scoring model SCORING_FUN3, whose implementation is stored in ETLIN.SOURCE_TABLE, and return the output in ascending order of the first output column:

```
SELECT *
  FROM v1 ANALYZE_TABLE(
    IMPLEMENTATION
      'PROVIDER=SAS;
       ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
       ROUTINE_SOURCE_NAME=SCORING_FUN3;')
  ORDER BY 1;
```

*nested-table-expression*

A fullselect in parentheses is called a *nested table expression*. The intermediate result table is the result of that fullselect. The columns of the result do not need unique names, but a column with a non-unique name cannot be explicitly referenced. If LATERAL is specified, the fullselect can include correlated references to results columns of table references specified to the left of the nested table expression. If the nested table expression involves data from a federated data source, a continue-handler can be specified to tolerate certain error conditions from the data source.

An expression in the select list of a nested table expression that is referenced within, or is the target of, a data change statement within a fullselect is valid only when it does not include:

- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has external action
- An OLAP function

If a view is referenced directly in, or as the target of a nested table expression in a data change statement within a FROM clause, the view must meet either of the following conditions:

- Be symmetric (have WITH CHECK OPTION specified)
- Satisfy the restriction for a WITH CHECK OPTION view

If the target of a data change statement within a FROM clause is a nested table expression, the following restrictions apply:

- Modified rows are not requalified
- WHERE clause predicates are not reevaluated
- ORDER BY or FETCH FIRST operations are not redone

A nested table expression can be used in the following situations:

- In place of a view to avoid creating the view (when general use of the view is not required)
- When the required intermediate result table is based on host variables

*data-change-table-reference*

A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A *data-change-table-reference* can be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a *select-statement*, a SELECT INTO statement, or a common table expression. A *data-change-table-reference* can be specified as the only table reference in the only fullselect in a SET Variable statement (SQLSTATE 428FL). The target table or view of the data change statement is considered to be a table or view that is referenced in the query;

therefore, the authorization ID of the query must have SELECT privilege on that target table or view. A *data-change-table-reference* clause cannot be specified in a view definition, materialized query table definition, or FOR statement (SQLSTATE 428FL).

The target of the UPDATE, DELETE, or INSERT statement cannot be a temporary view defined in a common table expression (SQLSTATE 42807) or a nickname (SQLSTATE 25000).

Expressions in the select list of a view or fullselect as target of a data change statement in a *table-reference* can be selected only if OLD TABLE is specified or the expression does not include the following elements (SQLSTATE 428G6):

- A subquery
- A function that reads or modifies SQL data
- A function is that is non-deterministic or has an external action
- An OLAP function
- A NEXT VALUE FOR *sequence* reference

**FINAL TABLE**
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they exist at the completion of the data change statement. If there are AFTER triggers or referential constraints that result in further operations on the table that is the target of the SQL data change statement, an error is returned (SQLSTATE 560C6). If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned (SQLSTATE 428G3).

**NEW TABLE**
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement before the application of referential constraints and AFTER triggers. Data in the target table at the completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

**OLD TABLE**
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they existed before the application of the data change statement.

*(searched-update-statement)*
Specifies a searched UPDATE statement. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated references to columns outside of the UPDATE statement.

*(searched-delete-statement)*
Specifies a searched DELETE statement. A WHERE clause in the DELETE statement cannot contain correlated references to columns outside of the DELETE statement.

*(insert-statement)*
Specifies an INSERT statement. A fullselect in the INSERT statement cannot contain correlated references to columns outside of the fullselect of the INSERT statement.

The content of the intermediate result table for a *data-change-table-reference* is determined when the cursor opens. The intermediate result table contains all manipulated rows, including all the columns in the specified target table or

view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

*table-function-reference*

In general, a table function, together with its argument values, can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view. Each *function-name* together with the types of its arguments, specified as a table reference must resolve to an existing table function at the application server. There are, however, some special considerations which apply.

- **Table function column names**: Unless alternative column names are provided following the *correlation-name*, the column names for the table function are those specified in the RETURNS or RETURNS GENERIC TABLE clause of the CREATE FUNCTION statement. This is analogous to the names of the columns of a table, which are defined in the CREATE TABLE statement.
- **Table function resolution**: The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions) that are used in a statement.
- **Table function arguments**: As with scalar function arguments, table function arguments can generally be any valid SQL expression. The following examples are valid syntax:

```
Example 1:  SELECT c1
            FROM TABLE( tf1('Zachary') ) AS z
            WHERE c2 = 'FLORIDA';

Example 2:  SELECT c1
            FROM TABLE( tf2 (:hostvar1, CURRENT DATE) ) AS z;

Example 3:  SELECT c1
            FROM t
            WHERE c2 IN
                     (SELECT c3 FROM
                      TABLE( tf5(t.c4) ) AS z  -- correlated reference
                     )                         -- to previous FROM clause
Example 4:  SELECT c1
            FROM TABLE( tf6('abcd') )               -- tf6 is a generic
                AS z (c1 int, c2 varchar(100)) -- java table function
```

- **Table functions that modify SQL data**: Table functions that are specified with the MODIFIES SQL DATA option can be used only as the last table reference in a *select-statement*, *common-table-expression*, or RETURN statement that is a subselect, a SELECT INTO, or a *row-fullselect* in a SET statement. Only one table function is allowed in one FROM clause, and the table function arguments must be correlated to all other table references in the subselect (SQLSTATE 429BL). The following examples have valid syntax for a table function with the MODIFIES SQL DATA property:

```
Example 1:  SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 2:  SELECT c1
            FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) ) AS z

Example 3:  SET var =
            (SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z
```

```
Example 4:  RETURN SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 5:  WITH v1(c1) AS
            (SELECT c1
            FROM TABLE( tfmod(:hostvar1) ) AS z)
            SELECT c1
            FROM v1, t1 WHERE v1.c1 = t1.c1

Example 6: SELECT z.*
           FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) )
           AS z (col1 int)
```

*collection-derived-table*

A *collection-derived-table* can be used to convert the elements of an array into values of a column in separate rows. If WITH ORDINALITY is specified, an extra column of data type INTEGER is appended. This column contains the position of the element in the array. The columns can be referenced in the select list and the in rest of the subselect by using the names specified for the columns in the correlation-clause. The *collection-derived-table* clause can be used only in a context where arrays are supported (SQLSTATE 42887). See the "UNNEST table function" for details.

*xmltable-expression*

An *xmltable-expression* specifies an invocation of the built-in XMLTABLE function which determines the intermediate result table. See XMLTABLE for more information.

*joined-table*

A *joined-table* specifies an intermediate result set that is the result of one or more join operations. For more information, see "joined-table" on page 307.

*period-specification*

A *period-specification* identifies an intermediate result table consisting of the rows of the referenced table where the period matches the specification. A *period-specification* can be specified following the name of a temporal table or the name of a view. The same period name must not be specified more than once for the same table reference (SQLSTATE 428HY). The rows of the table reference are derived by application of the period specifications.

If the table is a system-period temporal table and a *period-specification* for the SYSTEM_TIME period is not specified, the table reference includes all current rows and does not include any historical rows of the table. If the table is an application-period temporal table and a *period-specification* for the BUSINESS_TIME period is not specified, the table reference includes all rows of the table. If the table is a bitemporal table and a *period-specification* is not specified for both SYSTEM_TIME and BUSINESS_TIME, the table reference includes all current rows of the table and does not include any historical rows of the table.

If the table reference is a single-view-reference, the rows of the view reference are derived by application of the period specifications to all of the temporal tables accessed when computing the result table of the view. If the view does not access any temporal table, then the *period-specification* has no effect on the result table of the view. If *period-specification* is used, the view definition or any view definitions referenced when computing the result table of the view must not include any references to compiled SQL functions or external functions with a data access indication other than NO SQL (SQLSTATE 428HY).

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a value other than the null value, then a *period-specification* that references SYSTEM_TIME must not be specified for the table reference or view reference, unless the value in effect for the SYSTIMESENSITIVE bind option is NO (SQLSTATE 428HY).

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a value other than the null value, then a period specification that references BUSINESS_TIME must not be specified for the table reference or view reference, unless the value in effect for the BUSTIMESENSITIVE bind option is NO (SQLSTATE 428HY).

**FOR SYSTEM_TIME**

Specifies that the SYSTEM_TIME period is used for the *period-specification*. If the clause is specified following a *table-name*, the table must be a system-period temporal table (SQLSTATE 428HY). FOR SYSTEM_TIME must not be specified if the value of the CURRENT TEMPORAL SYSTEM_TIME special register is not the null value and the SYSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

**FOR BUSINESS_TIME**

Specifies that the BUSINESS_TIME period is used for the *period-specification*. If the clause is specified following a *table-name*, BUSINESS_TIME must be a period defined in the table (SQLSTATE 4274M). FOR BUSINESS_TIME must not be specified if the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value and the BUSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

*value*, *value1*, **and** *value2*

The *value*, *value1*, and *value2* expressions return the null value or a value of one of the following built-in data types (SQLSTATE 428HY): a DATE, a TIMESTAMP, or a character string that is not a CLOB or DBCLOB. If the argument is a character string, it must be a valid character string representation of a timestamp or a date (SQLSTATE 22007). For the valid formats of string representations of timestamp values, see the section "String representations of datetime values" in the topic "Datetime values".

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable (host-variable, SQL parameter, SQL variable, transition variable)
- Parameter marker
- Scalar function whose arguments are supported operands (user-defined functions and non-deterministic functions cannot be used)
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

**AS OF** *value*

Specifies that the table reference includes each row for which the value of the begin column for the specified period is less than or equal to *value*, and the value of the end column for the period is greater than *value*. If *value* is the null value, the table reference is an empty table.

*Example:* The following query returns the insurance coverage information for insurance policy number 100 on August 31, 2010.

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
  AS OF '2010-08-31' WHERE policy_id = '100'
```

**FROM** *value1* **TO** *value2*

Specifies that the table reference includes rows that exist for the period specified from *value1* to *value2*. A row is included in the table reference if the value of the begin column for the specified period in the row is less than *value2*, and the value of the end column for the specified period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than or equal to *value2*. If *value1* or *value2* is the null value, the table reference is an empty table.

*Example:* The following query returns the insurance coverage information for insurance policy 100, during the year 2009 (from January 1, 2009 at 12:00 AM until before January 1, 2010).

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
  FROM '2009-01-01' TO '2010-01-01' WHERE policy_id = '100'
```

**BETWEEN** *value1* **AND** *value2*

Specifies that the table reference includes rows in which the specified period overlaps at any point in time between *value1* and *value2*. A row is included in the table reference if the value of the begin column for the specified period in the row is less than or equal to *value2* and the value of the end column for the specified period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than *value2*. If *value1* is equal to *value2*, the expression is equivalent to AS OF *value1*. If *value1* or *value2* is the null value, the table reference is an empty table.

*Example:* The following query returns the insurance coverage information for insurance policy number 100, during the year 2008 (between January 1, 2008 and December 31, 2008 inclusive).

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
  BETWEEN '2008-01-01' AND '2008-12-31' WHERE policy_id = '100'
```

Following are syntax alternatives for *period-specification* clauses:

- AS OF TIMESTAMP can be specified in place of FOR SYSTEM_TIME AS OF
- VERSIONS BETWEEN TIMESTAMP can be specified in place of FOR SYSTEM_TIME BETWEEN

*correlation-clause*

The exposed names of all table references must be unique. An exposed name is:

- A *correlation-name*
- A *table-name* that is not followed by a *correlation-name*
- A *view-name* that is not followed by a *correlation-name*
- A *nickname* that is not followed by a *correlation-name*
- An *alias-name* that is not followed by a *correlation-name*

If a *correlation-clause* clause does not follow a *function-name* reference, *xmltable-expression* expression, nested table expression, or *data-change-table-reference* reference, or if a *typed-correlation-clause* clause does not follow a *function-name* reference, then there is no exposed name for that table reference.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *nickname*, *function-name* reference, *xmltable-expression*, nested table expression, or *data-change-table-reference*. Any qualified reference to a column must use the exposed name. If the same table name, view, or nickname is specified twice, at least one specification must be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table, view or nickname. When a *correlation-name* is specified, *column-name*s can also be specified to give names to the columns of the table reference. If the *correlation-clause* does not include *column-name*s, the exposed column names are determined as follows:

- Column names of the referenced table, view, or nickname when the *table-reference* is a *table-name*, *view-name*, *nickname*, or *alias-name*
- Column names specified in the RETURNS clause of the CREATE FUNCTION statement when the *table-reference* is a *function-name* reference
- Column names specified in the COLUMNS clause of the *xmltable-expression* when the *table-reference* is an *xmltable-expression*
- Column names exposed by the fullselect when the *table-reference* is a *nested-table-expression*
- Column names from the target table of the data change statement, along with any defined INCLUDE columns when the *table-reference* is a *data-change-table-reference*

*typed-correlation-clause*

A *typed-correlation-clause* clause defines the appearance and contents of the table generated by a generic table function. This clause must be specified when the table-function-references is a generic table function and cannot be specified for any other table reference. The following *data-type* values are supported in generic table functions:

*Table 24. Data types supported in generic table functions*

| SQL column data type | Equivalent Java data type |
|---|---|
| SMALLINT | short |
| INTEGER | int |
| BIGINT | long |
| REAL | float |
| DOUBLE | double |
| DECIMAL(p,s) | java.math.BigDecimal |
| NUMERIC(p,s) | java.math.BigDecimal |
| CHAR(n) | java.lang.String |
| CHAR(n) FOR BIT DATA | COM.ibm.db2.app.Blob |
| VARCHAR(n) | java.lang.String |
| VARCHAR(n) FOR BIT DATA | COM.ibm.db2.app.Blob |
| GRAPHIC(n) | java.lang.String |
| VARGRAPHIC(n) | String |
| BLOB(n) | COM.ibm.db2.app.Blob |
| CLOB(n) | COM.ibm.db2.app.Clob |
| DBCLOB(n) | COM.ibm.db2.app.Clob |
| DATE | String |

*Table 24. Data types supported in generic table functions (continued)*

| SQL column data type | Equivalent Java data type |
|---|---|
| TIME | String |
| TIMESTAMP | String |
| XML AS CLOB(n) | COM.ibm.db2.jcc.DB2Xml |

*tablesample-clause*

The optional *tablesample-clause* can be used to obtain a random subset (a sample) of the rows from the specified *table-name*, rather than the entire contents of that *table-name*, for this query. This sampling is in addition to any predicates that are specified in the *where-clause*. Unless the optional REPEATABLE clause is specified, each execution of the query will usually yield a different sample, except in degenerate cases where the table is so small relative to the sample size that any sample must return the same rows. The size of the sample is controlled by the *numeric-expression1* in parentheses, representing an approximate percentage (P) of the table to be returned.

**TABLESAMPLE**

The method by which the sample is obtained is specified after the TABLESAMPLE keyword, and can be either BERNOULLI or SYSTEM. For both methods, the exact number of rows in the sample might be different for each execution of the query, but on average is approximately P percent of the table, before any predicates further reduce the number of rows.

The *table-name* must be a stored table. It can be a materialized query table (MQT) name, but not a subselect or table expression for which an MQT has been defined, because there is no guarantee that the database manager will route to the MQT for that subselect.

Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. Repeated accesses of a sampled table within a single execution of a query (such as in a nested-loop join or a correlated subquery) will return the same sample. More than one table can be sampled in a query.

**BERNOULLI**

BERNOULLI sampling considers each row individually. It includes each row in the sample with probability P/100 (where P is the value of *numeric-expression1*), and excludes each row with probability 1 - P/100, independently of the other rows. So if the *numeric-expression1* evaluated to the value 10, representing a ten percent sample, each row would be included with probability 0.1, and excluded with probability 0.9.

**SYSTEM**

SYSTEM sampling permits the database manager to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a *table-name* means that each page of *table-name* is included in the sample with probability P/100, and excluded with probability 1 - P/100. All rows on each page that is included qualify for the sample. SYSTEM sampling of a *table-name* generally executes much faster than BERNOULLI sampling, because fewer data pages are retrieved. However, SYSTEM sampling can often yield less accurate estimates for aggregate functions, such as SUM(SALES), especially if the rows of *table-name* are clustered on any columns referenced in that query. The optimizer might in certain circumstances decide that it is more efficient to

perform SYSTEM sampling as if it were BERNOULLI sampling. An example is when a predicate on *table-name* can be applied by an index and is much more selective than the sampling rate P.

*numeric-expression1*

The *numeric-expression1* specifies the size of the sample to be obtained from *table-name*, expressed as a percentage. It must be a constant numeric expression that cannot contain columns. The expression must evaluate to a positive number that is less than or equal to 100, but can be between 1 and 0. For example, a value of 0.01 represents one one-hundredth of a percent, meaning that 1 row in 10 000 is sampled, on average. A *numeric-expression1* that evaluates to 100 is handled as if the tablesample-clause were not specified. If *numeric-expression1* evaluates to the null value, or to a value that is greater than 100 or less than 0, an error is returned (SQLSTATE 2202H).

**REPEATABLE (***numeric-expression2***)**

It is sometimes desirable for sampling to be repeatable from one execution of the query to the next; for example, during regression testing or query debugging. This can be accomplished by specifying the REPEATABLE clause. The REPEATABLE clause requires the specification of a *numeric-expression2* in parentheses, which serves the same role as the seed in a random number generator. Adding the REPEATABLE clause to the tablesample-clause of any *table-name* ensures that repeated executions of that query (using the same value for *numeric-expression2*) return the same sample, assuming that the data itself has not been updated, reorganized, or repartitioned. To guarantee that the same sample of *table-name* is used across multiple queries, use of a global temporary table is recommended. Alternatively, the multiple queries can be combined into one query, with multiple references to a sample that is defined using the WITH clause.

- *Example 1:* Request a 10% Bernoulli sample of the Sales table for auditing purposes.

```
SELECT * FROM Sales
  TABLESAMPLE BERNOULLI(10)
```

- *Example 2:* Compute the total sales revenue in the Northeast region for each product category, using a random 1% SYSTEM sample of the Sales table. The semantics of SUM are for the sample itself, so to extrapolate the sales to the entire Sales table, the query must divide that SUM by the sampling rate (0.01).

```
SELECT SUM(Sales.Revenue) / (0.01)
  FROM Sales TABLESAMPLE SYSTEM(1)
  WHERE Sales.RegionName = 'Northeast'
  GROUP BY Sales.ProductCategory
```

- *Example 3:* Using the REPEATABLE clause, modify the previous query to ensure that the same (yet random) result is obtained each time the query is executed. The value of the constant enclosed by parentheses is arbitrary.

```
SELECT SUM(Sales.Revenue) / (0.01)
  FROM Sales TABLESAMPLE SYSTEM(1) REPEATABLE(3578231)
  WHERE Sales.RegionName = 'Northeast'
  GROUP BY Sales.ProductCategory
```

*continue-handler*

Certain errors that occur within a *nested-table-expression* can be tolerated, and instead of returning an error, the query can continue and return a result. This is referred to as an *error tolerant nested-table-expression*.

Specifying the RETURN DATA UNTIL clause will cause any rows that are returned from the fullselect before the indicated condition is encountered to make up the result set from the fullselect. This means that a partial result set (which can also be an empty result set) from the fullselect is acceptable as the result for the *nested-table-expression*.

The FEDERATED keyword restricts the condition to handle only errors that occur at a remote data source.

The condition can be specified as an SQLSTATE value, with a *string-constant* length of 5. You can optionally specify an SQLCODE value for each specified SQLSTATE value. For portable applications, specify SQLSTATE values as much as possible, because SQLCODE values are generally not portable across platforms and are not part of the SQL standard.

Only certain conditions can be tolerated. Errors that do not allow the rest of the query to be executed cannot be tolerated, and an error is returned for the whole query. The *specific-condition-value* might specify conditions that cannot actually be tolerated by the database manager, even if a specific SQLSTATE or SQLCODE value is specified, and for these cases, an error is returned.

A query or view containing an error tolerant *nested-table-expression* is read-only.

The fullselect of an error tolerant *nested-table-expression* is not optimized using materialized query tables.

*specific-condition-value*

The following SQLSTATE values and SQLCODE values have the potential, when specified, to be tolerated by the database manager:

- SQLSTATE 08001; SQLCODEs -1336, -30080, -30081, -30082
- SQLSTATE 08004
- SQLSTATE 42501
- SQLSTATE 42704; SQLCODE -204
- SQLSTATE 42720
- SQLSTATE 28000

## Correlated references in table-references

Correlated references can be used in nested table expressions or as arguments to table functions. The basic rule that applies for both of these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the LATERAL keyword must exist before the fullselect. The following examples have valid syntax:

```
Example 1:  SELECT t.c1, z.c5
            FROM t, TABLE( tf3(t.c2) ) AS z      -- t precedes tf3
            WHERE t.c3 = z.c4;                   -- in FROM, so t.c2
                                                 -- is known

Example 2:  SELECT t.c1, z.c5
            FROM t, TABLE( tf4(2 * t.c2) ) AS z  -- t precedes tf4
            WHERE t.c3 = z.c4;                   -- in FROM, so t.c2
                                                 -- is known

Example 3:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                 LATERAL (SELECT AVG(e.salary) AS avgsal,
```

```
            COUNT(*) AS empcount
         FROM employee e           -- department precedes nested
         WHERE e.workdept=d.deptno -- table expression and
        ) AS empinfo;              -- LATERAL is specified,
                                   -- so d.deptno is known
```

But the following examples are not valid:

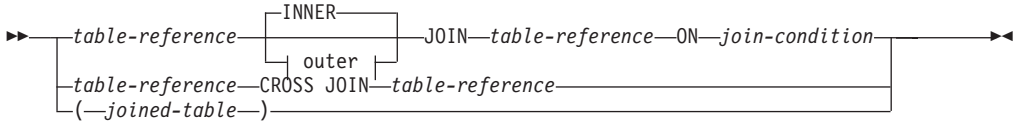
```
Example 4:  SELECT t.c1, z.c5
            FROM TABLE( tf6(t.c2) ) AS z, t  -- cannot resolve t in t.c2!
            WHERE t.c3 = z.c4;               -- compare to Example 1 above.

Example 5:  SELECT a.c1, b.c5
            FROM TABLE( tf7a(b.c2) ) AS a, TABLE( tf7b(a.c6) ) AS b
            WHERE a.c3 = b.c4;               -- cannot resolve b in b.c2!

Example 6:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                 (SELECT AVG(e.salary) AS avgsal,
                            COUNT(*) AS empcount
                    FROM employee e           -- department precedes nested
                    WHERE e.workdept=d.deptno -- table expression but
                   ) AS empinfo;              -- LATERAL is not specified,
                                              -- so d.deptno is unknown
```

## joined-table

A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: CROSS, INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.




**outer:**


Cross joins represent the cross product of the tables, where each row of the left table is combined with every row of the right table. Inner joins can be thought of as the cross product of the tables, keeping only the rows where the join condition is true. The result table might be missing rows from either or both of the joined tables. Outer joins include the inner join and preserve these missing rows. There are three types of outer joins:

- ***Left outer join*** includes rows from the left table that were missing from the inner join.
- ***Right outer join*** includes rows from the right table that were missing from the inner join.
- ***Full outer join*** includes rows from both the left and right tables that were missing from the inner join.

If a join-operator is not specified, INNER is implicit. The order in which multiple joins are performed can affect the result. Joins can be nested within other joins. The order of processing for joins is generally from left to right, but based on the position of the required join-condition. Parentheses are recommended to make the order of nested joins more readable. For example:

```
TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1
  RIGHT JOIN TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1
    ON TB1.C1=TB3.C1
```

is the same as:

```
(TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1)
  RIGHT JOIN (TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1)
    ON TB1.C1=TB3.C1
```

A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

A *join-condition* is a *search-condition*, except that:

- It cannot include any dereference operations or the DEREF function, where the reference value is other than the object identifier column
- Any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- Any function referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action
- It cannot include an XMLQUERY or XMLEXISTS expression

An error occurs if the join condition does not comply with these rules (SQLSTATE 42972).

Column references are resolved using the rules for resolution of column name qualifiers. The same rules that apply to predicates apply to join conditions.

### Join operations

A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of the JOIN operator of the *join-condition*. For all possible combinations of rows of T1 and T2, a row of T1 is paired with a row of T2 if the *join-condition* is true. When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. The execution might involve the generation of a null row. The null row of a table consists of a null value for each column of the table, regardless of whether the null values are allowed in the columns.

The following list summarizes the result of the join operations:

- The result of T1 CROSS JOIN T2 consists of all possible pairings of their rows.
- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T1, the concatenation of that row with the null row of T2. Null values are allowed in all columns derived from T2.

- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T2, the concatenation of that row with the null row of T1. Null values are allowed in all columns derived from T1.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. Null values are allowed in all columns derived from T1 and T2.

## where-clause

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the subselect.

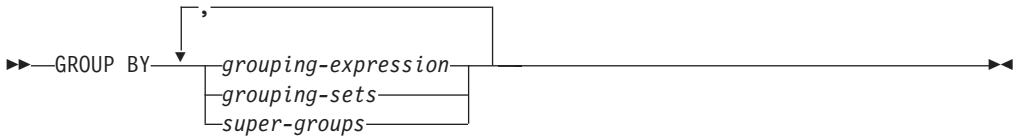

The *search-condition* must conform to the following rules:

- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R, and the results are used in the application of the *search-condition* to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references might be executed just once, whereas a subquery with a correlated reference might be executed once for each row.

## group-by-clause

The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.




In its simplest form, a GROUP BY clause contains a *grouping expression*. A grouping expression is an *expression* used in defining the grouping of R. Each expression or *column name* included in grouping-expression must unambiguously identify a column of R (SQLSTATE 42702 or 42703). A grouping expression cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any expression or function that is not deterministic or has an external action (SQLSTATE 42845).

**Note:** The following expressions, which do not contain an explicit column reference, can be used in a *grouping-expression* to identify a column of R:

- ROW CHANGE TIMESTAMP FOR *table-designator*

- ROW CHANGE TOKEN FOR *table-designator*
- RID_BIT or RID scalar function

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of *grouping-sets*, see "grouping-sets." For a description of *super-groups*, see "super-groups" on page 311.

The result of GROUP BY is a set of groups of rows. Each row in this result represents the set of rows for which the *grouping-expression* is equal. For grouping, all null values from a *grouping-expression* are considered equal.

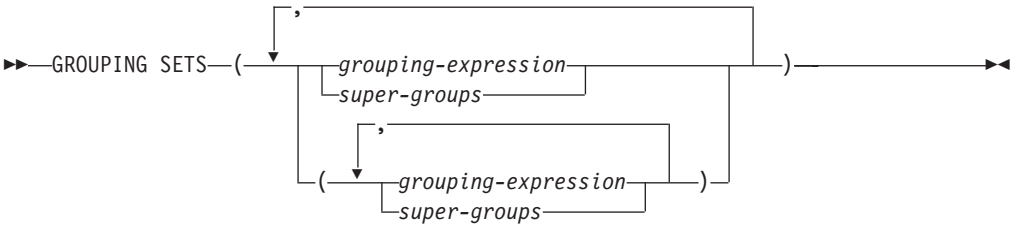
If a *grouping-expression* contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the number that is returned can be any of the representations of the number.

A *grouping-expression* can be used in a search condition in a HAVING clause, in an expression in a SELECT clause or in a *sort-key-expression* of an ORDER BY clause (see "order-by-clause" on page 316 for details). In each case, the reference specifies only one value for each group. For example, if the *grouping-expression* is *col1+col2*, then an allowed expression in the select list is *col1+col2+3*. Associativity rules for expressions disallow the similar expression, *3+col1+col2*, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, *3+(col1+col2)* is also allowed in the select list. If the concatenation operator is used, the *grouping-expression* must be used exactly as the expression was specified in the select list.

If the *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and might not all have the same length. In that case, a reference to the *grouping-expression* still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

As noted, there are some cases where the GROUP BY clause cannot refer directly to a column that is specified in the SELECT clause as an expression (scalar-fullselect, not deterministic or external action functions). To group using such an expression, use a nested table expression or a common table expression to first provide a result table with the expression as a column of the result. For an example using nested table expressions, see Example 9 in "Examples of subselect queries" on page 321.

## grouping-sets




A *grouping-sets* specification can be used to specify multiple grouping clauses in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple

subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a grouping-expression or a super-group. The groups can be computed with a single pass over the base table using *grouping-sets*.

A simple *grouping-expression* or the more complex forms of *super-groups* are supported by the *grouping-sets* specification. For a description of *super-groups*, see "super-groups."

Note that grouping sets are the fundamental building blocks for GROUP BY operations. A simple GROUP BY with a single column can be considered a grouping set with one element. For example:

```
GROUP BY a
```

is the same as

```
GROUP BY GROUPING SETS((a))
```

and

```
GROUP BY a,b,c
```

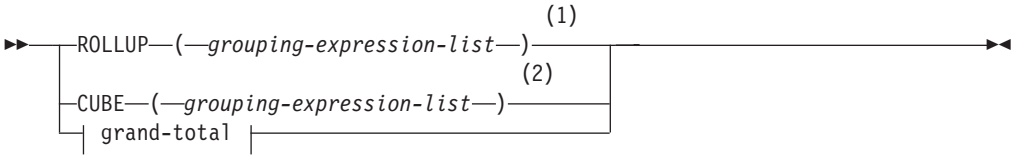
is the same as

```
GROUP BY GROUPING SETS((a,b,c))
```

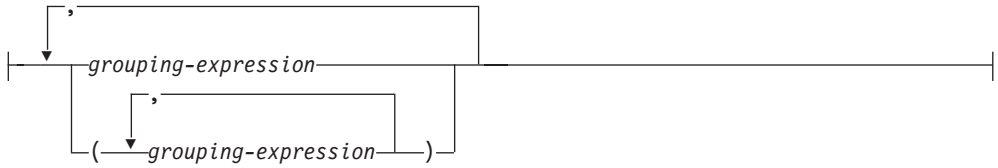
Non-aggregation columns from the select list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns.

The use of grouping sets is illustrated in Example 2 through Example 7 in "Examples of subselect queries with grouping sets, cube, and rollup queries" on page 325.

## super-groups




### grouping-expression-list:




### grand-total:

**Notes:**

1 Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH ROLLUP.

2 Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH CUBE.

**ROLLUP ( *grouping-expression-list* )**

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set containing *sub-total* rows in addition to the "regular" grouped rows. *Sub-total* rows are "super-aggregate" rows that contain further aggregates whose values are derived by applying the same aggregate functions that were used to obtain the grouped rows. These rows are called sub-total rows, because that is their most common use; however, any aggregate function can be used for the aggregation. For instance, MAX and AVG are used in Example 8 in "Examples of subselect queries with grouping sets, cube, and rollup queries" on page 325. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

A ROLLUP grouping is a series of *grouping-sets*. The general specification of a ROLLUP with *n* elements

```
GROUP BY ROLLUP(C1,C2,...,Cn-1,Cn)
```

is equivalent to

```
GROUP BY GROUPING SETS((C1,C2,...,Cn-1,Cn)
                       (C1,C2,...,Cn-1)
                       ...
                       (C1,C2)
                       (C1)
                       () )
```

Note that the *n* elements of the ROLLUP translate to *n*+1 grouping sets. Note also that the order in which the *grouping-expressions* is specified is significant for ROLLUP. For example, the following clause:

```
GROUP BY ROLLUP(a,b)
```

is equivalent to:

```
GROUP BY GROUPING SETS((a,b)
                       (a)
                       () )
```

Similarly, the following clause:

```
GROUP BY ROLLUP(b,a)
```

is equivalent to:

```
GROUP BY GROUPING SETS((b,a)
                       (b)
                       () )
```

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example 3 in "Examples of subselect queries with grouping sets, cube, and rollup queries" on page 325 illustrates the use of ROLLUP.

**CUBE ( *grouping-expression-list* )**

A *CUBE grouping* is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains "cross-tabulation" rows. *Cross-tabulation* rows are additional

"super-aggregate" rows that are not part of an aggregation with sub-totals. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

Similar to a ROLLUP, a CUBE grouping can also be thought of as a series of *grouping-sets*. In the case of a CUBE, all permutations of the cubed *grouping-expression-list* are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to 2**n* (2 to the power *n*) *grouping-sets*. For example, a specification of:

```
GROUP BY CUBE(a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (b,c)
                       (a)
                       (b)
                       (c)
                       () )
```

Note that the three elements of the CUBE translate into eight grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales_Person)' and 'CUBE (Sales_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. The use of CUBE is illustrated in Example 4 in "Examples of subselect queries with grouping sets, cube, and rollup queries" on page 325.

*grouping-expression-list*

A *grouping-expression-list* is used within a CUBE or ROLLUP clause to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expression*s.

For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However, the clause:

```
GROUP BY ROLLUP(Province, County, City)
```

results in unwanted subtotal rows for the County. In the clause:

```
GROUP BY ROLLUP(Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the required result. In other words, the two-element ROLLUP:

```
GROUP BY ROLLUP(Province, (County, City))
```

generates:

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province)
                       () )
```

and the three-element ROLLUP generates:

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province, County)
                       (Province)
                       () )
```

Example 2 in "Examples of subselect queries with grouping sets, cube, and rollup queries" on page 325 also utilizes composite column values.

**grand-total**
Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This can be separately specified with empty parentheses within the GROUPING SET clause. It can also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. Example 4 in "Examples of subselect queries with grouping sets, cube, and rollup queries" on page 325 uses the grand-total syntax.

## Combining grouping sets

This can be used to combine any of the types of GROUP BY clauses. When simple *grouping-expression* fields are combined with other groups, they are "appended" to the beginning of the resulting *grouping sets*. When ROLLUP or CUBE expressions are combined, they operate similarly to "multipliers" on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining *grouping-expression* elements acts as follows:

```
GROUP BY a, ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a) )
```

Or similarly,

```
GROUP BY a, b, ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b) )
```

Combining of *ROLLUP* elements acts as follows:

```
GROUP BY ROLLUP(a), ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a)
                       (b,c)
                       (b)
                       () )
```

Similarly,

```
GROUP BY ROLLUP(a), CUBE(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (a)
```

```
                        (b,c)
                        (b)
                        (c)
                        () )
```

Combining of *CUBE* and *ROLLUP* elements acts as follows:

```
GROUP BY CUBE(a,b), ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b)
                       (a,c,d)
                       (a,c)
                       (a)
                       (b,c,d)
                       (b,c)
                       (b)
                       (c,d)
                       (c)
                       () )
```

Similar to a simple *grouping-expression*, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```
GROUP BY a, ROLLUP(a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b)
                       (a) )
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that might be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

```
GROUP BY Region,
         ROLLUP(Sales_Person, WEEK(Sales_Date)),
         CUBE(YEAR(Sales_Date), MONTH (Sales_Date))
```

The column listed immediately to the right of GROUP BY is grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the GROUP BY clause results in a cube of MONTH within YEAR which is then rolled up within WEEK within Sales_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales_Person or WEEK(Sales_Date) so produces fewer rows than the clause:

```
GROUP BY ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
                 YEAR(Sales_Date), MONTH(Sales_Date) )
```

## having-clause

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the *search-condition* is true. R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered to be a single group with no grouping columns.

Each *column-name* in the search condition must satisfy one of the following conditions:

- Unambiguously identify a grouping column of R.
- Be specified within an aggregate function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each aggregate function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example 6 and Example 7 in "Examples of subselect queries" on page 321.

A correlated reference to a group of R must either identify a grouping column or be contained within an aggregate function.

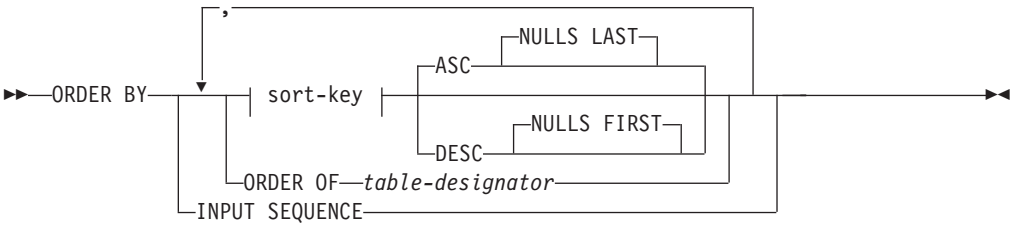
When HAVING is used without GROUP BY, the select list can only include column names when they are arguments to an aggregate function, correlated column references, global variables, host variables, literals, special registers, SQL variables, or SQL parameters.

**Note:** The following expressions can only be specified in a HAVING clause if they are contained within an aggregate function (SQLSTATE 42803):

- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID_BIT or RID scalar function

## order-by-clause

The ORDER BY clause specifies an ordering of the rows of the result table.




**sort-key:**


If a single sort specification (one *sort-key* with associated direction) is identified, the rows are ordered by the values of that sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified

sort specification, then by the values of the second identified sort specification, and so on. Each *sort-key* cannot have a data type of CLOB, DBCLOB, BLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907).

A named column in the select list can be identified by a *sort-key* that is a *simple-integer* or a *simple-column-name*. An unnamed column in the select list must be identified by an *simple-integer* or, in some cases, by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). A column is unnamed if the AS clause is not specified and it is derived from a constant, an expression with operators, or a function.

Ordering is performed in accordance with comparison rules. If an ORDER BY clause contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the ordering of the multiple representations of the same number is unspecified. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

*simple-column-name*
Usually identifies a column of the result table. In this case, *simple-column-name* must be the column name of a named column in the select list.

The *simple-column-name* can also identify a column name of a table, view, or nested table identified in the FROM clause if the query is a subselect. This includes columns defined as implicitly hidden. An error occurs in the following situations:

- If the subselect specifies DISTINCT in the select-clause (SQLSTATE 42822)
- If the subselect produces a grouped result and the *simple-column-name* is not a *grouping-expression* (SQLSTATE 42803)

Determining which column is used for ordering the result is described under "Column names in sort keys" in the "Notes" section.

*simple-integer*
Must be greater than 0 and not greater than the number of columns in the result table (SQLSTATE 42805). The integer *n* identifies the *n*th column of the result table.

*sort-key-expression*
An expression that is not simply a column name or an unsigned integer constant. The query to which ordering is applied must be a *subselect* to use this form of sort-key. The *sort-key-expression* cannot include a correlated scalar fullselect (SQLSTATE 42703) or a function with an external action (SQLSTATE 42845).

Any *column-name* within a *sort-key-expression* must conform to the rules described under "Column names in sort keys" in the "Notes" section.

There are a number of special cases that further restrict the expressions that can be specified.

- DISTINCT is specified in the SELECT clause of the subselect (SQLSTATE 42822).

  The sort-key-expression must match exactly with an expression in the select list of the subselect (scalar-fullselects are never matched).
- The subselect is grouped (SQLSTATE 42803).

  The sort-key-expression can:

  – be an expression in the select list of the subselect,

- include a *grouping-expression* from the GROUP BY clause of the subselect
- include an aggregate function, constant or host variable.

**ASC**
Uses the values of the column in ascending order. This is the default.

**DESC**
Uses the values of the column in descending order.

**ORDER OF** *table-designator*
Specifies that the same ordering used in *table-designator* applies to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Note that this form is not allowed in a fullselect (other than the degenerative form of a fullselect). For example, the following is not valid:

```
(SELECT C1 FROM T1
   ORDER BY C1)
UNION
SELECT C1 FROM T2
   ORDER BY ORDER OF T1
```

The following example *is* valid:

```
SELECT C1 FROM
   (SELECT C1 FROM T1
      UNION
    SELECT C1 FROM T2
    ORDER BY C1 ) AS UTABLE
ORDER BY ORDER OF UTABLE
```

**INPUT SEQUENCE**
Specifies that, for an INSERT statement, the result table will reflect the input order of ordered data rows. INPUT SEQUENCE ordering can only be specified if an INSERT statement is used in a FROM clause (SQLSTATE 428G4). See "table-reference" on page 289. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

## Notes

- **Column names in sort keys**:
  - The column name is qualified.

    The query must be a *subselect* (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.
  - The column name is unqualified.
    - The query is a subselect.

      If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it

must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).

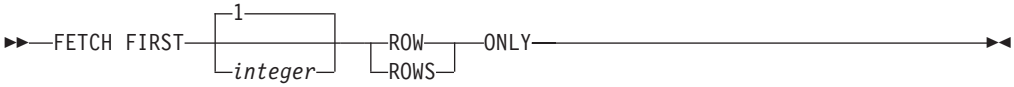
- The query is not a subselect (it includes set operations such as union, except or intersect).

  The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707), and this column is used to compute the value of the sort specification.

- **Limits**: The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list might result in the addition of the column or expression to the temporary table used for sorting. This might result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

## fetch-first-clause

The *fetch-first-clause* sets a maximum number of rows that can be retrieved.




The *fetch-first-clause* lets the database manager know that the application does not want to retrieve more than *integer* rows, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data (SQLSTATE 02000). The value of *integer* must be a positive integer (not zero).

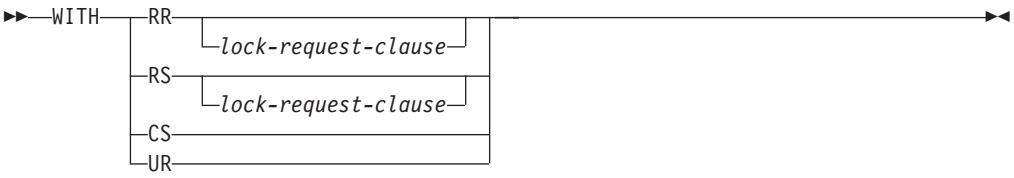
Use of the *fetch-first-clause* influences query optimization of the subselect or fullselect, based on the fact that at most *integer* rows will be retrieved. If both the fetch-first-clause is specified in the outermost fullselect and the optimize-for-clause is specified for the select statement, the database manager will use the *integer* from the optimize-for-clause to influence query optimization of the outermost fullselect.

Limiting the result table to the first *integer* rows can improve performance. The database manager will cease processing the query when it has determined the first *integer* rows. If both the *fetch-first-clause* and the *optimize-for-clause* are specified, the lower of the *integer* values from these clauses is used to influence the communications buffer size.

If the fullselect contains an SQL data change statement in the FROM clause, all the rows are modified regardless of the limit on the number of rows to fetch.

## isolation-clause (subselect query)

The optional *isolation-clause* specifies the isolation level at which the subselect or fullselect is run, and whether a specific type of lock is to be acquired.




- RR - Repeatable-Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

## lock-request-clause


The *lock-request-clause* applies only to queries and to positioning read operations within an insert, update, or delete operation. The insert, update, and delete operations themselves will run using locking determined by the database manager.

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold:

**SHARE**
Concurrent processes can acquire SHARE or UPDATE locks on the data.

**UPDATE**
Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

**EXCLUSIVE**
Concurrent processes cannot acquire a lock on the data.

*isolation-clause* **restrictions**:

- The *isolation-clause* is not supported for a CREATE TABLE, CREATE VIEW, or ALTER TABLE statement (SQLSTATE 42601).
- The *isolation-clause* cannot be specified for a subselect or fullselect that will cause trigger invocation, referential integrity scans, or MQT maintenance (SQLSTATE 42601).
- A subselect or fullselect cannot include a *lock-request-clause* if that subselect or fullselect references any SQL functions that are not declared with the option INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601).
- A subselect or fullselect that contains a *lock-request-clause* are not be eligible for MQT routing.
- If an *isolation-clause* is specified for a *subselect* or *fullselect* within the body of an SQL function, SQL method, or trigger, the clause is ignored and a warning is returned.
- If an *isolation-clause* is specified for a *subselect* or *fullselect* that is used by a scrollable cursor, the clause is ignored and a warning is returned.
- Neither *isolation-clause* nor *lock-request-clause* can be specified in the context where they will cause conflict isolation or lock intent on a common table

expression (SQLSTATE 42601). This restriction does not apply to aliases or base tables. The following examples create an isolation conflict on *a* and returns an error:

– View:

```
create view a as (...);
(select * from a with RR USE AND KEEP SHARE LOCKS)
UNION ALL
(select * from a with UR);
```

– Common table expression:

```
WITH a as (...)
(select * from a with RR USE AND KEEP SHARE LOCKS)
UNION ALL
(select * from a with UR);
```

- An *isolation-clause* cannot be specified in an XML context (SQLSTATE 2200M).

## Examples of subselect queries

The following examples illustrate the susbelect query.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

- *Example 2:* Join the EMP_ACT and EMPLOYEE tables, select all the columns from the EMP_ACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```
SELECT EMP_ACT.*, LASTNAME
  FROM EMP_ACT, EMPLOYEE
  WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
```

- *Example 3:* Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1930.

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
   FROM EMPLOYEE, DEPARTMENT
   WHERE WORKDEPT = DEPTNO
   AND YEAR(BIRTHDATE) < 1930
```

- *Example 4:* Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.

```
SELECT JOB, MIN(SALARY), MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY JOB
  HAVING COUNT(*) > 1
  AND MAX(SALARY) >= 27000
```

- *Example 5:* Select all the rows of EMP_ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
SELECT *
  FROM EMP_ACT
  WHERE EMPNO IN
           (SELECT EMPNO
               FROM EMPLOYEE
               WHERE WORKDEPT = 'E11')
```

- *Example 6:* From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                        FROM EMPLOYEE)
```

The subquery in the HAVING clause is executed once in this example.

- *Example 7:* Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

```
SELECT WORKDEPT, MAX(SALARY)
   FROM EMPLOYEE EMP_COR
    GROUP BY WORKDEPT
    HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                          FROM EMPLOYEE
                          WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

In contrast to Example 6, the subquery in the HAVING clause is executed for each group.

- *Example 8:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments.

  This query must first create a nested table expression (DINFO) to get the AVGSALARY and EMPCOUNT columns, and the DEPTNO column that is used in the WHERE clause.

```
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
 FROM EMPLOYEE THIS_EMP,
      (SELECT OTHERS.WORKDEPT AS DEPTNO,
                          AVG(OTHERS.SALARY) AS AVGSALARY,
                          COUNT(*) AS EMPCOUNT
        FROM EMPLOYEE OTHERS
        GROUP BY OTHERS.WORKDEPT
       ) AS DINFO
 WHERE THIS_EMP.JOB = 'SALESREP'
  AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Using a nested table expression for this case saves the processing resources of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the query, only the rows for the department of the sales representatives are considered by the view.

- *Example 9:* Display the average education level and salary for 5 random groups of employees.

  This query requires the use of a nested table expression to set a random value for each employee so that it can subsequently be used in the GROUP BY clause.

```
SELECT RANDID , AVG(EDLEVEL), AVG(SALARY)
  FROM ( SELECT EDLEVEL, SALARY, INTEGER(RAND()*5) AS RANDID
           FROM EMPLOYEE
       ) AS EMPRAND
  GROUP BY RANDID
```

- *Example 10:* Query the EMP_ACT table and return those project numbers that have an employee whose salary is in the top 10 of all employees.

```
SELECT EMP_ACT.EMPNO,PROJNO
   FROM EMP_ACT
   WHERE EMP_ACT.EMPNO IN
       (SELECT EMPLOYEE.EMPNO
        FROM EMPLOYEE
        ORDER BY SALARY DESC
        FETCH FIRST 10 ROWS ONLY)
```

- *Example 11:* Assuming that PHONES and IDS are two SQL variables with array values of the same cardinality, turn these arrays into a table with three columns (one for each array and one for the position), and one row per array element.

```
SELECT T.PHONE, T.ID, T.INDEX FROM UNNEST(PHONES, IDS)
  WITH ORDINALITY AS T(PHONE, ID, INDEX)
  ORDER BY T.INDEX
```

## Examples of subselect queries with joins

The following examples illustrate the use of joins in a subselect query.

- *Example 1:* This example illustrates the results of the various joins using tables J1 and J2. These tables contain rows as shown.

```
SELECT * FROM J1

W   X
--- ------
A       11
B       12
C       13

SELECT * FROM J2

Y   Z
--- ------
A       21
C       22
D       23
```

The following query does an inner join of J1 and J2 matching the first column of both tables.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
```

In this inner join example the row with column W='C' from J1 and the row with column Y='D' from J2 are not included in the result because they do not have a match in the other table. Note that the following alternative form of an inner join query produces the same result.

```
SELECT * FROM J1, J2 WHERE W=Y
```

The following left outer join will get back the missing row from J1 with nulls for the columns of J2. Every row from J1 is included.

```
SELECT * FROM J1 LEFT OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
B       12 -        -
C       13 C       22
```

The following right outer join will get back the missing row from J2 with nulls for the columns of J1. Every row from J2 is included.

```
SELECT * FROM J1 RIGHT OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
-        - D       23
```

The following full outer join will get back the missing rows from both J1 and J2 with nulls where appropriate. Every row from both J1 and J2 is included.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
-        - D       23
B       12 -        -
```

- *Example 2:* Using the tables J1 and J2 from the previous example, examine what happens when and additional predicate is added to the search condition.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=13

W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

The additional condition caused the inner join to select only 1 row compared to the inner join in Example 1.

Notice what the affect of this is on the full outer join.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=13

W   X      Y   Z
--- ------ --- ------
-        - A       21
C       13 C       22
-        - D       23
A       11 -        -
B       12 -        -
```

The result now has 5 rows (compared to 4 without the additional predicate) because there was only 1 row in the inner join and all rows of both tables must be returned.

The following query illustrates that placing the same additional predicate in WHERE clause has completely different results.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
   WHERE X=13

W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

The WHERE clause is applied after the intermediate result of the full outer join. This intermediate result is the same as the result of the full outer join query in Example 1. The WHERE clause is applied to this intermediate result and eliminates all but the row that has X=13. Choosing the location of a predicate when performing outer joins can have a significant affect on the results. Consider what happens if the predicate was X=12 instead of X=13. The following inner join returns no rows.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=12
```

Hence, the full outer join returns 6 rows, 3 from J1 with nulls for the columns of J2 and 3 from J2 with nulls for the columns of J1.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=12

W   X      Y   Z
--- ------ --- ------
-        - A       21
-        - C       22
```

```
-        - D       23
A       11 -        -
B       12 -        -
C       13 -        -
```

If the additional predicate is in the WHERE clause instead, 1 row is returned.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
   WHERE X=12

W   X      Y   Z
--- ------ --- ------
B       12 -        -
```

- *Example 3:* List every department with the employee number and last name of the manager, including departments without a manager.

```
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
  FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
       ON MGRNO = EMPNO
```

- *Example 4:* List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

```
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
  FROM EMPLOYEE E LEFT OUTER JOIN
                                        DEPARTMENT INNER JOIN EMPLOYEE M
       ON MGRNO = M.EMPNO
       ON E.WORKDEPT = DEPTNO
```

The inner join determines the last name for any manager identified in the DEPARTMENT table and the left outer join guarantees that each employee is listed even if a corresponding department is not found in DEPARTMENT.

## Examples of subselect queries with grouping sets, cube, and rollup queries

The following examples illustrate the grouping, cube, and rollup forms of subselect queries.

The queries in Example 1 through Example 4 use a subset of the rows in the SALES tables based on the predicate 'WEEK(SALES_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 LUCCHESSI                 3
         13           6 LUCCHESSI                 1
         13           6 LEE                       2
         13           6 LEE                       2
         13           6 LEE                       3
         13           6 LEE                       5
         13           6 GOUNOT                    3
         13           6 GOUNOT                    1
         13           6 GOUNOT                    7
         13           7 LUCCHESSI                 1
         13           7 LUCCHESSI                 2
         13           7 LUCCHESSI                 1
         13           7 LEE                       7
         13           7 LEE                       3
```

```
13           7 LEE                      7
13           7 LEE                      4
13           7 GOUNOT                   2
13           7 GOUNOT                  18
13           7 GOUNOT                   1
```

- *Example 1:* Here is a query with a basic GROUP BY clause over 3 columns:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

  This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
```

- *Example 2:* Produce the result based on two different grouping sets of rows from the SALES table.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS ( (WEEK(SALES_DATE), SALES_PERSON),
                         (DAYOFWEEK(SALES_DATE), SALES_PERSON))
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

  This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           - GOUNOT                   32
         13           - LEE                      33
         13           - LUCCHESSI                 8
          -           6 GOUNOT                   11
          -           6 LEE                      12
          -           6 LUCCHESSI                 4
          -           7 GOUNOT                   21
          -           7 LEE                      21
          -           7 LUCCHESSI                 4
```

  The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

- *Example 3:* If you use the 3 distinct columns involved in the grouping sets of Example 2 and perform a ROLLUP, you can see grouping sets for (WEEK,DAY_WEEK,SALES_PERSON), (WEEK, DAY_WEEK), (WEEK) and grand total.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

  This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - -                        73
          -           - -                        73
```

- *Example 4:* If you run the same query as Example 3 only replace ROLLUP with CUBE, you can see additional grouping sets for (WEEK,SALES_PERSON), (DAY_WEEK,SALES_PERSON), (DAY_WEEK), (SALES_PERSON) in the result.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY CUBE ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - GOUNOT                   32
         13           - LEE                      33
         13           - LUCCHESSI                 8
         13           - -                        73
          -           6 GOUNOT                   11
          -           6 LEE                      12
          -           6 LUCCHESSI                 4
          -           6 -                        27
          -           7 GOUNOT                   21
          -           7 LEE                      21
          -           7 LUCCHESSI                 4
          -           7 -                        46
          -           - GOUNOT                   32
          -           - LEE                      33
          -           - LUCCHESSI                 8
          -           - -                        73
```

- *Example 5:* Obtain a result set which includes a grand-total of selected rows from the SALES table together with a group of rows aggregated by SALES_PERSON and MONTH.

```
SELECT SALES_PERSON,
       MONTH(SALES_DATE) AS MONTH,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( (SALES_PERSON, MONTH(SALES_DATE)),
                         ()
                       )
ORDER BY SALES_PERSON, MONTH
```

This results in:

```
SALES_PERSON    MONTH       UNITS_SOLD
--------------- ----------- -----------
GOUNOT                    3          35
GOUNOT                    4          14
GOUNOT                   12           1
LEE                       3          60
LEE                       4          25
LEE                      12           6
LUCCHESSI                 3           9
LUCCHESSI                 4           4
LUCCHESSI                12           1
-                         -         155
```

- *Example 6:* This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.
  - *Example 6-1:*

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
ORDER BY WEEK, DAY_WEEK
```

    results in:

```
WEEK        DAY_WEEK    UNITS_SOLD
----------- ----------- -----------
         13           6          27
         13           7          46
         13           -          73
         14           1          31
         14           2          43
         14           -          74
         53           1           8
         53           -           8
          -           -         155
```

  - *Example 6-2:*

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( MONTH(SALES_DATE), REGION );
ORDER BY MONTH, REGION
```

    results in:

```
MONTH       REGION          UNITS_SOLD
----------- --------------- -----------
          3 Manitoba                 22
          3 Ontario-North             8
          3 Ontario-South            34
          3 Quebec                   40
          3 -                       104
          4 Manitoba                 17
          4 Ontario-North             1
          4 Ontario-South            14
          4 Quebec                   11
          4 -                        43
         12 Manitoba                  2
         12 Ontario-South             4
         12 Quebec                    2
         12 -                         8
          - -                       155
```

  - *Example 6-3:*

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) ),
                         ROLLUP( MONTH(SALES_DATE), REGION )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

```
WEEK        DAY_WEEK    MONTH       REGION          UNITS_SOLD
----------- ----------- ----------- --------------- -----------
         13           6           - -                        27
         13           7           - -                        46
         13           -           - -                        73
         14           1           - -                        31
         14           2           - -                        43
         14           -           - -                        74
         53           1           - -                         8
         53           -           - -                         8
          -           -           3 Manitoba                 22
          -           -           3 Ontario-North             8
          -           -           3 Ontario-South            34
          -           -           3 Quebec                   40
          -           -           3 -                       104
          -           -           4 Manitoba                 17
          -           -           4 Ontario-North             1
          -           -           4 Ontario-South            14
          -           -           4 Quebec                   11
          -           -           4 -                        43
          -           -          12 Manitoba                  2
          -           -          12 Ontario-South             4
          -           -          12 Quebec                    2
          -           -          12 -                         8
          -           -           - -                       155
          -           -           - -                       155
```

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Observe how the use of ORDER BY has affected the results:

- In the first grouped set, week 53 has been repositioned to the end.
- In the second grouped set, month 12 has now been positioned to the end and the regions now display in alphabetic order.
- Null values are sorted high.

- *Example 7:* In queries that perform multiple ROLLUPs in a single pass (such as Example 6-3) you might want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called GROUP) which indicates the *origin* of each row in the result set. *Origin* means which one of the two grouping sets produced the row in the result set.

  *Step 1:* Introduce a way of "generating" new data values, using a query which selects from a VALUES clause (which is an alternative form of a fullselect). This query shows how a table can be derived called "X" having 2 columns "R1" and "R2" and 1 row of data.

  ```
  SELECT R1,R2
  FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
  ```

  results in:

  ```
  R1      R2
  ------- -------
  GROUP 1 GROUP 2
  ```

*Step 2:* Form the cross product of this table "X" with the SALES table. This add columns "R1" and "R2" to every row.

```
SELECT R1, R2, WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SALES AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
```

This add columns "R1" and "R2" to every row.

*Step 3:* Now these columns can be combined with the grouping sets to include these columns in the rollup analysis.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                         DAYOFWEEK(SALES_DATE))),
                       (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

```
R1      R2      WEEK     DAY_WEEK  MONTH     REGION          UNITS_SOLD
------- ------- -------- --------- --------- --------------- -----------
GROUP 1 -             13         6         - -                          27
GROUP 1 -             13         7         - -                          46
GROUP 1 -             13         -         - -                          73
GROUP 1 -             14         1         - -                          31
GROUP 1 -             14         2         - -                          43
GROUP 1 -             14         -         - -                          74
GROUP 1 -             53         1         - -                           8
GROUP 1 -             53         -         - -                           8
-       GROUP 2        -         -         3 Manitoba                   22
-       GROUP 2        -         -         3 Ontario-North               8
-       GROUP 2        -         -         3 Ontario-South              34
-       GROUP 2        -         -         3 Quebec                     40
-       GROUP 2        -         -         3 -                         104
-       GROUP 2        -         -         4 Manitoba                   17
-       GROUP 2        -         -         4 Ontario-North               1
-       GROUP 2        -         -         4 Ontario-South              14
-       GROUP 2        -         -         4 Quebec                     11
-       GROUP 2        -         -         4 -                          43
-       GROUP 2        -         -        12 Manitoba                    2
-       GROUP 2        -         -        12 Ontario-South               4
-       GROUP 2        -         -        12 Quebec                      2
-       GROUP 2        -         -        12 -                           8
-       GROUP 2        -         -         - -                         155
GROUP 1 -              -         -         - -                         155
```

*Step 4:* Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```
SELECT COALESCE(R1,R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
```

```
                        DAYOFWEEK(SALES_DATE))),
                       (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
  ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;
```

results in:

```
GROUP   WEEK        DAY_WEEK    MONTH       REGION          UNITS_SOLD
------- ----------- ----------- ----------- --------------- -----------
GROUP 1          13           6           - -                          27
GROUP 1          13           7           - -                          46
GROUP 1          13           -           - -                          73
GROUP 1          14           1           - -                          31
GROUP 1          14           2           - -                          43
GROUP 1          14           -           - -                          74
GROUP 1          53           1           - -                           8
GROUP 1          53           -           - -                           8
GROUP 1           -           -           - -                         155
GROUP 2           -           -           3 Manitoba                   22
GROUP 2           -           -           3 Ontario-North               8
GROUP 2           -           -           3 Ontario-South              34
GROUP 2           -           -           3 Quebec                     40
GROUP 2           -           -           3 -                         104
GROUP 2           -           -           4 Manitoba                   17
GROUP 2           -           -           4 Ontario-North               1
GROUP 2           -           -           4 Ontario-South              14
GROUP 2           -           -           4 Quebec                     11
GROUP 2           -           -           4 -                          43
GROUP 2           -           -          12 Manitoba                    2
GROUP 2           -           -          12 Ontario-South               4
GROUP 2           -           -          12 Quebec                      2
GROUP 2           -           -          12 -                           8
GROUP 2           -           -           - -                         155
```

- *Example 8:* The following example illustrates the use of various aggregate functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```
  SELECT MONTH(SALES_DATE) AS MONTH,
         REGION,
         SUM(SALES) AS UNITS_SOLD,
         MAX(SALES) AS BEST_SALE,
         CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
  FROM SALES
  GROUP BY CUBE(MONTH(SALES_DATE),REGION)
  ORDER BY MONTH, REGION
```

This results in:

```
MONTH       REGION          UNITS_SOLD  BEST_SALE   AVG_UNITS_SOLD
----------- --------------- ----------- ----------- --------------
          3 Manitoba                 22           7           3.14
          3 Ontario-North             8           3           2.67
          3 Ontario-South            34          14           4.25
          3 Quebec                   40          18           5.00
          3 -                       104          18           4.00
          4 Manitoba                 17           9           5.67
          4 Ontario-North             1           1           1.00
          4 Ontario-South            14           8           4.67
          4 Quebec                   11           8           5.50
          4 -                        43           9           4.78
         12 Manitoba                  2           2           2.00
         12 Ontario-South             4           3           2.00
         12 Quebec                    2           1           1.00
         12 -                         8           3           1.60
          - Manitoba                 41           9           3.73
          - Ontario-North             9           3           2.25
          - Ontario-South            52          14           4.00
          - Quebec                   53          18           4.42
          - -                       155          18           3.87
```

# Chapter 34. Cursors

To allow applications to retrieve a set of rows, SQL uses a mechanism called a cursor.

## Using a cursor to retrieve multiple rows

Using cursors in your applications to retrieve a set of rows requires that you issue SQL statements to declare cursors, open cursors, fetch cursors, and close cursors.

### About this task

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

### Procedure

To process a cursor:

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

### What to do next

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

## DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is not an executable statement and cannot be dynamically prepared. When invoked using the command line processor, additional options can be specified. For more information, refer to “Using command line SQL statements and XQuery statements”.

## Authorization

The term “SELECT statement of the cursor” is used to specify the authorization rules. The SELECT statement of the cursor is one of the following statements:

- The prepared select-statement identified by *statement-name*
- The specified *select-statement*

The privileges held by the authorization ID of the statement must include the privileges necessary to execute the *select-statement*. See the Authorization section in "SQL queries".

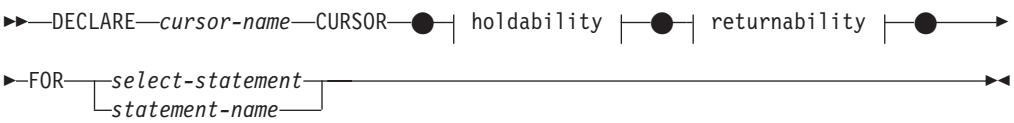
If *statement-name* is specified:

- The authorization ID of the statement is the runtime authorization ID.
- The authorization check is performed when the SELECT-statement is prepared.
- The cursor cannot be opened unless the SELECT-statement is in a prepared state.

If *select-statement* is specified:

- GROUP privileges are not checked.
- The authorization ID of the statement is the authorization ID specified during program preparation.

## Syntax




**holdability:**


**returnability:**


## Description

*cursor-name*
Specifies the name of the cursor created when the source program is run. The name must not be the same as the name of another cursor declared in the source program. The cursor must be opened before use.

**WITHOUT HOLD or WITH HOLD**
Specifies whether or not the cursor should be prevented from being closed as a consequence of a commit operation.

**WITHOUT HOLD**
Does not prevent the cursor from being closed as a consequence of a commit operation. This is the default.

**WITH HOLD**
Maintains resources across multiple units of work. The effect of the WITH HOLD cursor attribute is as follows:

- For units of work ending with COMMIT:
  - Open cursors defined WITH HOLD remain open. The cursor is positioned before the next logical row of the results table.

    If a DISCONNECT statement is issued after a COMMIT statement for a connection with WITH HOLD cursors, the held cursors must be explicitly closed or the connection will be assumed to have performed work (simply by having open WITH HELD cursors even though no SQL statements were issued) and the DISCONNECT statement will fail.
  - All locks are released, except locks protecting the current cursor position of open WITH HOLD cursors. The locks held include the locks on the table, and for parallel environments, the locks on rows where the cursors are currently positioned. Locks on packages and dynamic SQL sections (if any) are held.
  - Valid operations on cursors defined WITH HOLD immediately following a COMMIT request are:
    - FETCH: Fetches the next row of the cursor.
    - CLOSE: Closes the cursor.
  - UPDATE and DELETE CURRENT OF CURSOR are valid only for rows that are fetched within the same unit of work.
  - LOB locators are freed.
  - The set of rows modified by:
    - A data change statement
    - Routines that modify SQL data embedded within open WITH HOLD cursors

    is committed.
- For units of work ending with ROLLBACK:
  - All open cursors are closed.
  - All locks acquired during the unit of work are released.
  - LOB locators are freed.
- For special COMMIT case:
  - Packages can be recreated either explicitly, by binding the package, or implicitly, because the package has been invalidated and then dynamically recreated the first time it is referenced. All held cursors are closed during package rebind. This might result in errors during subsequent execution.

**WITHOUT RETURN or WITH RETURN**
Specifies whether or not the result table of the cursor is intended to be used as a result set that will be returned from a procedure.

**WITHOUT RETURN**
Specifies that the result table of the cursor is not intended to be used as a result set that will be returned from a procedure.

**WITH RETURN**
Specifies that the result table of the cursor is intended to be used as a result set that will be returned from a procedure. WITH RETURN is relevant only if the DECLARE CURSOR statement is contained with the source code for a procedure. In other cases, the precompiler might accept the clause, but it has no effect.

Within an SQL procedure, cursors declared using the WITH RETURN clause that are still open when the SQL procedure ends, define the result sets from the SQL procedure. All other open cursors in an SQL procedure are closed when the SQL procedure ends. Within an external procedure (one not defined using LANGUAGE SQL), the default for all cursors is WITH RETURN TO CALLER. Therefore, all cursors that are open when the procedure ends will be considered result sets. Cursors that are returned from a procedure cannot be declared as scrollable cursors.

**TO CALLER**
Specifies that the cursor can return a result set to the caller. For example, if the caller is another procedure, the result set is returned to that procedure. If the caller is a client application, the result set is returned to the client application.

**TO CLIENT**
Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function, method, or trigger called the procedure either directly or indirectly, result sets cannot be returned to the client and the cursor will be closed after the procedure finishes.

*select-statement*
Identifies the SELECT statement of the cursor. The *select-statement* must not include parameter markers, but can include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program.

*statement-name*
The SELECT statement of the cursor is the prepared SELECT statement identified by the *statement-name* when the cursor is opened. The *statement-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of the source program.

For an explanation of prepared SELECT statements, see “PREPARE”.

## Examples

*Example 1:* The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPARTMENT
  WHERE ADMRDEPT = 'A00';
```

*Example 2:* Assume that the EMPLOYEE table has been altered to add a generated column, WEEKLYPAY, that calculates the weekly pay based on the yearly salary. Declare a cursor to retrieve the system-generated column value from a row to be inserted.

```
EXEC SQL DECLARE C2 CURSOR FOR
  SELECT E.WEEKLYPAY
  FROM NEW TABLE
```

```
(INSERT INTO EMPLOYEE
 (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL, SALARY)
 VALUES('000420', 'Peter', 'U', 'Bender', 16, 31842) AS E;
```

# OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

## Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, some options cannot be specified. For more information, refer to "Using command line SQL statements and XQuery statements".

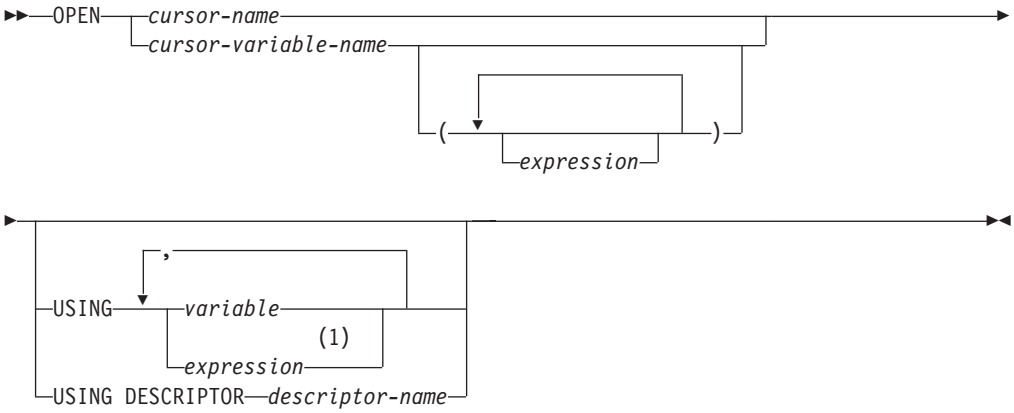
## Authorization

If a global variable is referenced, the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

Group privileges are not considered because this statement cannot be dynamically prepared.

## Syntax




**Notes:**

1 An expression other than a variable can only be used in compiled compound statements.

## Description

*cursor-name*

Names a cursor that is defined in a DECLARE CURSOR statement that was stated earlier in the program. If *cursor-name* identifies a cursor in an SQL procedure declared as WITH RETURN TO CLIENT that is already in the open

state, the existing open cursor becomes a result set cursor that is no longer accessible using cursor-name and a new cursor is opened that becomes accessible using cursor-name. Otherwise, when the OPEN statement is executed, the cursor identified by *cursor-name* must be in the closed state.

The DECLARE CURSOR statement must identify a SELECT statement, in one of the following ways:

- Including the SELECT statement in the DECLARE CURSOR statement
- Including a *statement-name* that names a prepared SELECT statement.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers, global variables, or PREVIOUS VALUE expressions specified in the SELECT statement, and the current values of any host variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement, and a temporary table may be created to hold them; or they may be derived during the execution of subsequent FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty, the state of the cursor is effectively "after the last row".

*cursor-variable-name*

Names a cursor variable. The value of the cursor variable must not be null (SQLSTATE 34000). A cursor variable that is directly or indirectly assigned a cursor value constructor can be used only in an OPEN statement that is in the same scope as the assignment (SQLSTATE 51044). If the cursor value constructor assigned to the cursor variable specified a *statement-name*, the OPEN statement must be in the same scope where that *statement-name* was explicitly or implicitly declared (SQLSTATE 51044).

When the OPEN statement is executed, the underlying cursor of the cursor variable must be in the closed state. The result table of the underlying cursor is derived by evaluating the SELECT statement or dynamic statement associated with the cursor variable. The evaluation uses the current values of any special registers, global variables, or PREVIOUS VALUE expressions specified in the SELECT statement, and the current values of any variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement, and a temporary table may be created to hold them; or they may be derived during the execution of subsequent FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty, the state of the cursor is effectively "after the last row".

An OPEN statement using a *cursor-variable-name* can only be used within a compound SQL (compiled) statement.

*( expression, ... )*

Specifies the arguments associated with the named parameters of a parameterized cursor variable. The *cursor-value-constructor* assigned to the cursor variable must include a list of parameters with the same number of parameters as the number of arguments specified (SQLSTATE 07006 or 07004). The data type and value of the *n*th expression must be assignable to the *n*th parameter (SQLSTATE 07006 or 22018).

**USING**

Introduces the values that are substituted for the parameter markers or variables in the statement of the cursor. For an explanation of parameter markers, see "PREPARE".

If a *statement-name* is specified in the DECLARE CURSOR statement or the cursor value constructor associated with the cursor variable that includes parameter markers, USING must be used. If the prepared statement does not include parameter markers, USING is ignored.

If a *select-statement* is specified in the DECLARE CURSOR statement or the non-parameterized cursor value constructor associated with the cursor variable, USING may be used to override the variable values.

**variable**

Identifies a variable or a host structure declared in the program in accordance with the rules for declaring variables and host variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Where appropriate, locator variables and file reference variables can be provided as the source of values for parameter markers.

**expression**

Specifies values to associate with parameter markers using expressions. An OPEN statement that specifies expressions in the USING clause can only be used within a compound SQL (compiled) statement (SQLSTATE 42601). The number of expressions must be the same as the number of parameter markers in the prepared statement (SQLSTATE 07001). The *n*th expression corresponds to the *n*th parameter marker in the prepared statement. The data type and value of the nth expression must be assignable to the type associated with the nth parameter marker (SQLSTATE 07006).

## Rules

- When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by its corresponding host variable. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker.
- Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:
  - V must be compatible with the target.
  - If V is a string, its length (excluding trailing blanks for strings that are not long strings) must not be greater than the length attribute of the target.
  - If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
  - If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

  When the SELECT statement of the cursor is evaluated, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6), and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.
- The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of

the cursor is part of the DECLARE CURSOR statement or the non-parameterized cursor value constructor associated with the cursor variable. In this case the OPEN statement is executed as if each host variable in the SELECT statement were a parameter marker, except that the attributes of the target variables are the same as the attributes of the host variables in the SELECT statement. The effect is to override the values of the host variables in the SELECT statement of the cursor with the values of the host variables specified in the USING clause. A variable value override must not be used when opening a parameterized cursor variable since the SELECT statement will not include any other variables.

- SQL data change statements and routines that modify SQL data embedded in the cursor definition are completely executed, and the result set is stored in a temporary table when the cursor opens. If statement execution is successful, the SQLERRD(3) field contains the sum of the number of rows that qualified for insert, update, and delete operations. If an error occurs during execution of an OPEN statement involving a cursor that contains a data change statement within a fullselect, the results of that data change statement are rolled back.

  Explicit rollback of an OPEN statement, or rollback to a savepoint before an OPEN statement, closes the cursor. If the cursor definition contains a data change statement within the FROM clause of a fullselect, the results of the data change statement are rolled back.

  Changes to rows in a table that is targeted by a data change statement nested within a SELECT statement or a SELECT INTO statement are processed when the cursor opens, and are not undone if an error occurs during a fetch operation against that cursor.

## Examples

*Example 1:* Write the embedded statements in a COBOL program that will:

1. Define a cursor C1 that is to be used to retrieve all rows from the DEPARTMENT table for departments that are administered by (ADMRDEPT) department 'A00'.
2. Place the cursor C1 before the first row to be fetched.

```
EXEC SQL  DECLARE C1 CURSOR FOR
               SELECT DEPTNO, DEPTNAME, MGRNO
                 FROM DEPARTMENT
                 WHERE ADMRDEPT = 'A00'
END-EXEC.


EXEC SQL  OPEN C1
END-EXEC.
```

*Example 2:* Code an OPEN statement to associate a cursor DYN_CURSOR with a dynamically defined select-statement in a C program. Assuming two parameter markers are used in the predicate of the select-statement, two host variable references are supplied with the OPEN statement to pass integer and varchar(64) values between the application and the database. (The related host variable definitions, PREPARE statement, and DECLARE CURSOR statement are also shown in this example.)

```
EXEC SQL  BEGIN DECLARE SECTION;
  static short    hv_int;
  char            hv_vchar64[65];
  char            stmt1_str[200];
EXEC SQL  END DECLARE SECTION;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;
```

```
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING :hv_int, :hv_vchar64;
```

*Example 3:* Code an OPEN statement as in example 2, but in this case the number and data types of the parameter markers in the WHERE clause are not known.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char    stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING DESCRIPTOR :sqlda;
```

*Example 4:* Create a procedure that does the following operations:

1. Assigns a cursor to the output cursor variable
2. Opens the cursor

```
CREATE PROCEDURE PROC1 (OUT P1 CURSOR)LANGUAGE SQL
BEGIN
SET P1=CURSOR FOR SELECT DEPTNO, DEPTNAME, MGRNO FROM DEPARTMENT WHERE ADMRDEPT='A00'; --
OPEN P1; --
END;
```

# FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to target variables.

## Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, the syntax following *cursor-name* is optional and different from the SQL syntax. For more information, refer to "Using command line SQL statements and XQuery statements".

## Authorization

For each global variable used as a *cursor-variable-name* or in the expression for an *array-index*, the privileges held by the authorization ID of the statement must include one of the following:

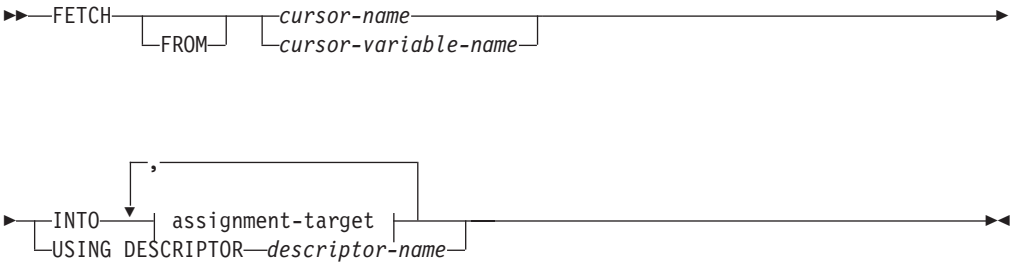
- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

For each global variable used as an *assignment-target*, the privileges held by the authorization ID of the statement must include one of the following:

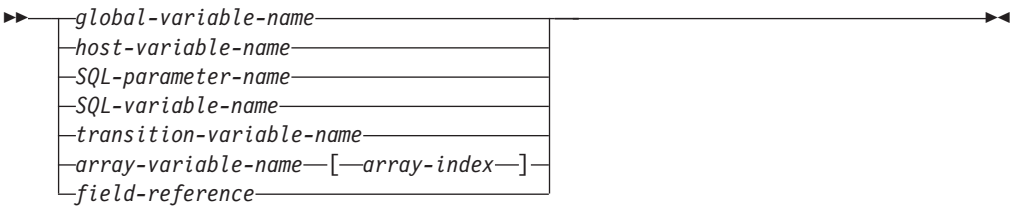
- WRITE privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

For the authorization required to use a cursor, see "DECLARE CURSOR".

## Syntax




### assignment-target




## Description

*cursor-variable-name*
Identifies the cursor to be used in the fetch operation. The cursor-variable-name must identify a cursor variable that is in scope. When the FETCH statement is executed, the underlying cursor of the *cursor-variable-name* must be in the open state. A FETCH statement using a *cursor-variable-name* can only be used within a compound SQL (compiled) statement.

**INTO** *assignment-target*

Identifies one or more targets for the assignment of output values. The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. Each assignment to an *assignment-target* is made in sequence through the list. If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to targets. Any values that have already been assigned to targets remain assigned.

When the data type of every *assignment-target* is not a row type, then the value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of *assignment-targets* is less than the number of result column values.

If the data type of an *assignment-target* is a row type, then there must be exactly one *assignment-target* specified (SQLSTATE 428HR), the number of columns must match the number of fields in the row type, and the data types of the columns of the fetched row must be assignable to the corresponding fields of the row type (SQLSTATE 42821).

If the data type of an *assignment-target* is an array element, then there must be exactly one *assignment-target* specified.

*global-variable-name*
Identifies the global variable that is the assignment target.

*host-variable-name*
Identifies the host variable that is the assignment target. For LOB output

values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

*SQL-parameter-name*
Identifies the parameter that is the assignment target.

*SQL-variable-name*
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

*transition-variable-name*
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value.

*array-variable-name*
Identifies an SQL variable, SQL parameter, or global variable of an array type.

*[array-index]*
An expression that specifies which element in the array will be the target of the assignment. For an ordinary array, the *array-index* expression must be assignable to INTEGER (SQLSTATE 428H1) and cannot be the null value. Its value must be between 1 and the maximum cardinality defined for the array (SQLSTATE 2202E). For an associative array, the *array-index* expression must be assignable to the index data type of the associative array (SQLSTATE 428H1) and cannot be the null value.

*field-reference*
Identifies the field within a row type value that is the assignment target. The *field-reference* must be specified as a qualified *field-name* where the qualifier identifies the row value in which the field is defined.

**USING DESCRIPTOR** *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host variables.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence.

If LOB or structured type result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table).

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

The *n*th variable described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of each variable must be compatible with its corresponding column.

Each assignment to a variable is made according to specific rules. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLDA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

### Examples

- *Example 1:* In this C example, the FETCH statement fetches the results of the SELECT statement into the program variables `dnum`, `dname`, and `mnum`. When no more rows remain to be fetched, the not found condition is returned.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT
    WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {
    EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
}

EXEC SQL CLOSE C1;
```

- *Example 2:* This FETCH statement uses an SQLDA.

```
FETCH CURS USING DESCRIPTOR :sqlda3
```

## CLOSE

The CLOSE statement closes a cursor. If a result table was created when the cursor was opened, that table is destroyed.

### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, some options cannot be specified. For more information, refer to “Using command line SQL statements and XQuery statements”.

### Authorization

If a global variable is referenced, the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

For the authorization required to use a cursor, see “DECLARE CURSOR”.

### Syntax

## Description

*cursor-name*
Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

*cursor-variable-name*
Identifies the cursor to be closed. The *cursor-variable-name* must identify a cursor variable. When the CLOSE statement is executed, the underlying cursor of *cursor-variable-name* must be in the open state (SQLSTATE 24501). A CLOSE statement using *cursor-variable-name* can only be used within a compound SQL (compiled) statement.

**WITH RELEASE**
The release of all locks that have been held for the cursor is attempted. Note that not all of the locks are necessarily released; these locks may be held for other operations or activities.

## Example

A cursor is used to fetch one row at a time into the C program variables `dnum`, `dname`, and `mnum`. Finally, the cursor is closed. If the cursor is reopened, it is again located at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM TDEPT
  WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {                                              .
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
     .
     .
}
EXEC SQL CLOSE C1;
```

# Chapter 35. Transactions

A transaction is a recoverable unit of work, or a group of SQL statements that can be treated as one atomic operation. This means that all the operations within the group are guaranteed to be completed (committed) or undone (rolled back), as if they were a single operation.

## COMMIT

The COMMIT statement terminates a unit of work and commits the database changes that were made by that unit of work.

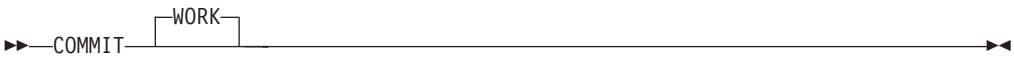
### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

None required.

### Syntax




### Description

The unit of work in which the COMMIT statement is executed is terminated and a new unit of work is initiated. All changes made by the following statements executed during the unit of work are committed: ALTER, COMMENT, CREATE, DROP, GRANT, LOCK TABLE, REVOKE, SET INTEGRITY, SET Variable, and the data change statements (INSERT, DELETE, MERGE, UPDATE), including those nested in a query.

The following statements, however, are not under transaction control and changes made by them are independent of the COMMIT statement:

- SET CONNECTION
- SET PASSTHRU

  **Note:** Although the SET PASSTHRU statement is not under transaction control, the passthru session initiated by the statement is under transaction control.
- SET SERVER OPTION
- Assignments to updatable special registers

All locks acquired by the unit of work subsequent to its initiation are released, except necessary locks for open cursors that are declared WITH HOLD. All open cursors not defined WITH HOLD are closed. Open cursors defined WITH HOLD remain open, and the cursor is positioned before the next logical row of the result table. (A FETCH must be performed before a positioned UPDATE or DELETE

statement is issued.) All LOB locators are freed. Note that this is true even when the locators are associated with LOB values retrieved via a cursor that has the WITH HOLD property.

Dynamic SQL statements prepared in a package bound with the `KEEPDYNAMIC YES` option are kept in the SQL context after a COMMIT statement. This is the default behavior. The statement might be implicitly prepared again, as a result of DDL operations that are rolled back within the unit of work. Inactive dynamic SQL statements prepared in a package bound with `KEEPDYNAMIC NO` are removed from the SQL context after a COMMIT. The statement must be prepared again before it can be executed in a new transaction.

All savepoints set within the transaction are released.

The following statements behave differently than other data definition language (DDL) and data control language (DCL) statements. Changes made by these statements do not take effect until the statement is committed, even for the current connection that issues the statement. Only one of these statements can be issued by any application at a time, and only one of these statements is allowed within any one unit of work. Each statement must be followed by a COMMIT or a ROLLBACK statement before another one of these statements can be issued.

- CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
- CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
- CREATE WORK ACTION, ALTER WORK ACTION, or DROP (WORK ACTION)
- CREATE WORK CLASS, ALTER WORK CLASS, or DROP (WORK CLASS)
- CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
- GRANT (Workload Privileges) or REVOKE (Workload Privileges)

## Notes

- It is strongly recommended that each application process explicitly ends its unit of work before terminating. If the application program ends normally without a COMMIT or ROLLBACK statement then the database manager attempts a commit or rollback depending on the application environment.
- For information about the impact of COMMIT on cached dynamic SQL statements, see “EXECUTE”.
- For information about potential impacts of COMMIT on created temporary tables, see “CREATE GLOBAL TEMPORARY TABLE”.
- For information about potential impacts of COMMIT on declared temporary tables, see “DECLARE GLOBAL TEMPORARY TABLE”.
- The following dynamic SQL statements may be active during COMMIT:
  - Open WITH HOLD cursor
  - COMMIT statement
  - CALL statements under which the COMMIT statement was executed

## Example

Commit alterations to the database made since the last commit point.

```
COMMIT WORK
```

# ROLLBACK

The ROLLBACK statement is used to back out of the database changes that were made within a unit of work or a savepoint.

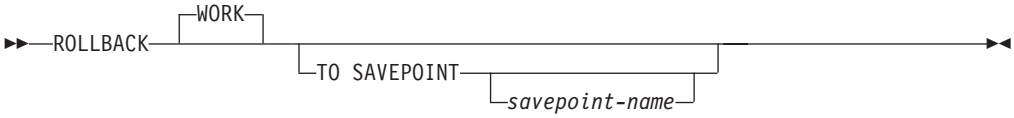
## Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

## Authorization

None required.

## Syntax




## Description

The unit of work in which the ROLLBACK statement is executed is terminated and a new unit of work is initiated. All changes made to the database during the unit of work are backed out.

The following statements, however, are not under transaction control, and changes made by them are independent of the ROLLBACK statement:

- SET CONNECTION
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE
- SET PASSTHRU (Although the SET PASSTHRU statement is not under transaction control, the passthru session initiated by the statement is under transaction control.)
- SET SERVER OPTION
- A SET statement that sets an updatable special register

The generation of sequence and identity values is not under transaction control. Values generated and consumed by the *nextval-expression* or by inserting rows into a table that has an identity column are independent of issuing the ROLLBACK statement. Also, issuing the ROLLBACK statement does not affect the value returned by the *prevval-expression*, nor the IDENTITY_VAL_LOCAL function.

Modification of the values of global variables is not under transaction control. ROLLBACK statements do not affect the values assigned to global variables.

**TO SAVEPOINT**
Specifies that a partial rollback (ROLLBACK TO SAVEPOINT) is to be performed. If no savepoint is active in the current savepoint level (see the "Rules" section in the description of the SAVEPOINT statement), an error is returned (SQLSTATE 3B502). After a successful rollback, the savepoint continues to exist, but any nested savepoints are released and no longer exist.

The nested savepoints, if any, are considered to have been rolled back and then released as part of the rollback to the current savepoint. If a *savepoint-name* is not provided, rollback occurs to the most recently set savepoint within the current savepoint level.

If this clause is omitted, the ROLLBACK statement rolls back the entire transaction. Furthermore, savepoints within the transaction are released.

*savepoint-name*
Specifies the savepoint that is to be used in the rollback operation. The specified *savepoint-name* cannot begin with 'SYS' (SQLSTATE 42939). After a successful rollback operation, the named savepoint continues to exist. If the savepoint name does not exist, an error (SQLSTATE 3B001) is returned. Data and schema changes made since the savepoint was set are undone.

## Notes

- All locks held are released on a ROLLBACK of the unit of work. All open cursors are closed. All LOB locators are freed.
- Executing a ROLLBACK statement does not affect either the SET statements that change special register values or the RELEASE statement.
- If the program terminates abnormally, the unit of work is implicitly rolled back.
- Statement caching is affected by the rollback operation.
- The impact on cursors resulting from a ROLLBACK TO SAVEPOINT depends on the statements within the savepoint
  - If the savepoint contains DDL on which a cursor is dependent, the cursor is marked invalid. Attempts to use such a cursor results in an error (SQLSTATE 57007).
  - Otherwise:
    - If the cursor is referenced in the savepoint, the cursor remains open and is positioned before the next logical row of the result table. (A FETCH must be performed before a positioned UPDATE or DELETE statement is issued.)
    - Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).
- Dynamic SQL statements prepared in a package bound with the `KEEPDYNAMIC YES` option are kept in the SQL context after a ROLLBACK statement. The statement might be implicitly prepared again, as a result of DDL operations that are rolled back within the unit of work.
- Inactive dynamic SQL statements prepared in a package bound with `KEEPDYNAMIC NO` are removed from the SQL context after a rollback operation. The statement must be prepared again before it can be executed in a new transaction.
- The following dynamic SQL statements may be active during ROLLBACK:
  - ROLLBACK statement
  - CALL statements under which the ROLLBACK statement was executed
- A ROLLBACK TO SAVEPOINT operation will drop any created temporary tables created within the savepoint. If a created temporary table is modified within the savepoint and that table has been defined as not logged, then all rows in the table are deleted.
- A ROLLBACK TO SAVEPOINT operation will drop any declared temporary tables declared within the savepoint. If a declared temporary table is modified within the savepoint and that table has been defined as not logged, then all rows in the table are deleted.
- All locks are retained after a ROLLBACK TO SAVEPOINT statement.

- All LOB locators are preserved following a ROLLBACK TO SAVEPOINT operation.

### Example

Delete the alterations made since the last commit point or rollback.

```
ROLLBACK WORK
```

## SAVEPOINT

Use the SAVEPOINT statement to set a savepoint within a transaction.

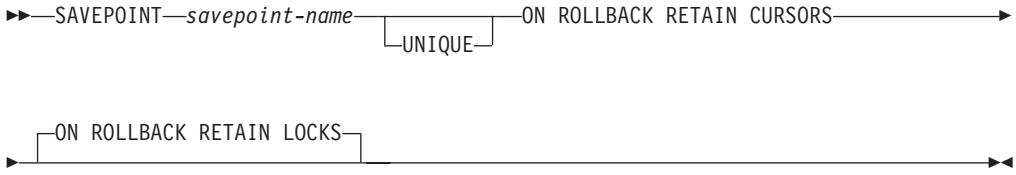
### Invocation

This statement can be imbedded in an application program (including a procedure) or issued interactively. It is an executable statement that can be dynamically prepared.

### Authorization

None required.

### Syntax




### Description

*savepoint-name*
Specifies the name of a savepoint. The specified *savepoint-name* cannot begin with 'SYS' (SQLSTATE 42939). If a savepoint by this name has already been defined as UNIQUE within this savepoint level, an error is returned (SQLSTATE 3B501).

**UNIQUE**
Specifies that the application does not intend to reuse this savepoint name while the savepoint is active within the current savepoint level. If *savepoint-name* already exists within this savepoint level, an error is returned (SQLSTATE 3B501).

**ON ROLLBACK RETAIN CURSORS**
Specifies system behavior upon rollback to this savepoint with respect to open cursor statements processed after the SAVEPOINT statement. This clause indicates that, whenever possible, the cursors are unaffected by a rollback to savepoint operation. For situations where the cursors are affected by the rollback to savepoint, see “ROLLBACK”.

**ON ROLLBACK RETAIN LOCKS**
Specifies system behavior upon rollback to this savepoint with respect to locks acquired after the setting of the savepoint. Locks acquired since the savepoint are not tracked, and are not rolled back (released) upon rollback to the savepoint.

## Rules

- Savepoint-related statements must not be used within trigger definitions (SQLSTATE 42987).
- A new savepoint level starts when one of the following events occurs:
  - A new unit of work (UOW) starts.
  - A procedure defined with the NEW SAVEPOINT LEVEL clause is called.
  - An atomic compound SQL statement starts.
- A savepoint level ends when the event that caused its creation is finished or removed. When a savepoint level ends, all savepoints contained within it are released. Any open cursors, DDL actions, or data modifications are inherited by the parent savepoint level (that is, the savepoint level within which the one that just ended was created), and are subject to any savepoint-related statements issued against the parent savepoint level.
- The following rules apply to actions within a savepoint level:
  - Savepoints can only be referenced within the savepoint level in which they are established. You cannot release, destroy, or roll back to a savepoint established outside of the current savepoint level.
  - All active savepoints established within the current savepoint level are automatically released when the savepoint level ends.
  - The uniqueness of savepoint names is only enforced within the current savepoint level. The names of savepoints that are active in other savepoint levels can be reused in the current savepoint level without affecting those savepoints in other savepoint levels.

## Notes

- Once a SAVEPOINT statement has been issued, insert, update, or delete operations on nicknames are not allowed.
- Omitting the UNIQUE clause specifies that *savepoint-name* can be reused within the savepoint level by another savepoint. If a savepoint of the same name already exists within the savepoint level, the existing savepoint is destroyed and a new savepoint with the same name is created at the current point in processing. The new savepoint is considered to be the last savepoint established by the application. Note that the destruction of a savepoint through the reuse of its name by another savepoint simply destroys that one savepoint and does not release any savepoints established after the destroyed savepoint. These subsequent savepoints can only be released by means of the RELEASE SAVEPOINT statement, which releases the named savepoint and all savepoints established after the named savepoint.
- If the UNIQUE clause is specified, *savepoint-name* can only be reused after an existing savepoint with the same name has been released.
- Within a savepoint, if a utility, SQL statement, or DB2 command performs intermittent commits during processing, the savepoint will be implicitly released.
- If the SET INTEGRITY statement is rolled back within the savepoint, dynamically prepared statement names are still valid, although the statement might be implicitly prepared again.
- If inserts are buffered (that is, the application was precompiled with the INSERT BUF option), the buffer will be flushed when SAVEPOINT, ROLLBACK, or RELEASE TO SAVEPOINT statements are issued.

## Example

Perform a rollback operation for nested savepoints. First, create a table named DEPARTMENT. Insert a row before starting SAVEPOINT1; insert another row and start SAVEPOINT2; then, insert a third row and start SAVEPOINT3.

```
CREATE TABLE DEPARTMENT (
  DEPTNO   CHAR(6),
  DEPTNAME VARCHAR(20),
  MGRNO    INTEGER)

  INSERT INTO DEPARTMENT VALUES ('A20', 'MARKETING', 301)

  SAVEPOINT SAVEPOINT1 ON ROLLBACK RETAIN CURSORS

  INSERT INTO DEPARTMENT VALUES ('B30', 'FINANCE', 520)

  SAVEPOINT SAVEPOINT2 ON ROLLBACK RETAIN CURSORS

  INSERT INTO DEPARTMENT VALUES ('C40', 'IT SUPPORT', 430)

  SAVEPOINT SAVEPOINT3 ON ROLLBACK RETAIN CURSORS

  INSERT INTO DEPARTMENT VALUES ('R50', 'RESEARCH', 150)
```

At this point, the DEPARTMENT table exists with rows A20, B30, C40, and R50. If you now issue:

```
ROLLBACK TO SAVEPOINT SAVEPOINT3
```

row R50 is no longer in the DEPARTMENT table. If you then issue:

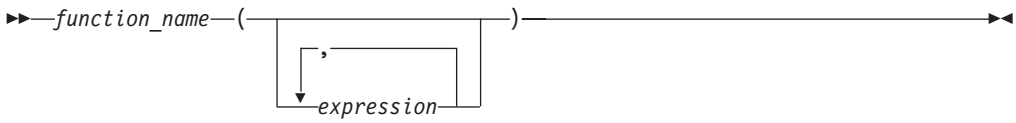
```
ROLLBACK TO SAVEPOINT SAVEPOINT1
```

the DEPARTMENT table still exists, but the rows inserted since SAVEPOINT1 was established (B30 and C40) are no longer in the table.

# Chapter 36. Invoking user-defined functions

Use the proper syntax to invoke user-defined functions so that the proper function is selected.

Each reference to a function, whether it is a UDF, or a built-in function, contains the following syntax:




In the preceding syntax diagram, `function_name` can be either an unqualified or a qualified function name. The arguments can number from 0 to 90 and are expressions. Examples of some components that can compose expressions are the following:

- a column name, qualified or unqualified
- a constant
- a host variable
- a special register
- a parameter marker

The position of the arguments is important and must conform to the function definition for the semantics to be correct. Both the position of the arguments and the function definition must conform to the function body itself. DB2 database systems do not attempt to shuffle arguments to better match a function definition, and do not understand the semantics of the individual function parameters.

Use of column names in UDF argument expressions requires that the table references that contain the columns have proper scope. For table functions referenced in a join and using any argument involving columns from another table or table function, the referenced table or table function must precede the table function containing the reference in the FROM clause.

In order to use parameter markers in functions you cannot simply code the following:

```
BLOOP(?)
```

Because the function selection logic does not know what data type the argument might turn out to be, it cannot resolve the reference. You can use the CAST specification to provide a type for the parameter marker. For example, INTEGER, and then the function selection logic can proceed:

```
BLOOP(CAST(? AS INTEGER))
```

Some valid examples of function invocations are:

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CHAR_COLUMN CONCAT USER, 1, 0, 0, 1)
```

```
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT FUNCTION PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
SYSFUN.FLOOR(AVG(EMP.SALARY))
SYSFUN.AVG(SYSFUN.FLOOR(EMP.SALARY))
SYSIBM.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
SQRT((SELECT SUM(length*length)
     FROM triangles
     WHERE id= 'J522'
     AND legtype <> 'HYP'))
```

If any of these functions are table functions, the syntax to reference them is slightly different than presented previously. For example, if PABLO.BLOOP is a table function, to properly reference it, use:

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```

## Invoking scalar functions or methods

The invocation of built-in scalar functions, user-defined scalar-functions and methods is very similar. Scalar functions and methods can only be invoked where expressions are supported within an SQL statement.

### Before you begin

- For built-in functions, SYSIBM must be in the CURRENT PATH special register. SYSIBM is in CURRENT PATH by default.
- For user-defined scalar functions, the function must have been created in the database using either the CREATE FUNCTION or CREATE METHOD statement.
- For external user-defined scalar functions, the library or class file associated with the function must be in the location specified by the EXTERNAL clause of the CREATE FUNCTION or CREATE METHOD statement.
- To invoke a user-defined function or method, a user must have EXECUTE privilege on the function or method. If the function or method is to be used by all users, the EXECUTE privilege on the function or method can be granted to PUBLIC.

### Procedure

To invoke a scalar UDF or method:

Include a reference to it within an expression contained in an SQL statement where it is to process one or more input values. Functions and methods can be invoked anywhere that an expression is valid. Examples of where a scalar UDF or method can be referenced include the select-list of a query or in a VALUES clause.

### Example

For example, suppose that you have created a user-defined scalar function called TOTAL_SAL that adds the base salary and bonus together for each employee row in the EMPLOYEE table.

```
CREATE FUNCTION TOTAL_SAL
  (SALARY DECIMAL(9,2), BONUS DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  LANGUAGE SQL
```

```
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SALARY+BONUS
```

The following is a SELECT statement that makes use of TOTAL_SAL:

```
SELECT LASTNAME, TOTAL_SAL(SALARY, BONUS) AS TOTAL
  FROM EMPLOYEE
```

## Invoking user-defined table functions

Once the user-defined table function is written and registered with the database, you can invoke it in the FROM clause of a SELECT statement.

### Before you begin

- The table function must have been created in the database by executing the CREATE FUNCTION.
- For external user-defined table functions, the library or class file associated with the function must be in the location specified by the EXTERNAL clause of the CREATE FUNCTION.
- To invoke a user-defined table function a user must have EXECUTE privilege on the function.

### Procedure

To invoke a user-defined table function, reference the function in the FROM clause of an SQL statement where it is to process a set of input values. The reference to the table function must be preceded by the TABLE clause and be contained in brackets.
For example, the following CREATE FUNCTION statement defines a table function that returns the employees in a specified department number.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO VARCHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
    SELECT EMPNO, LASTNAME, FIRSTNME FROM EMPLOYEE
      WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

The following is a SELECT statement that makes use of DEPTEMPLOYEES:

```
SELECT EMPNO, LASTNAME, FIRSTNAME FROM TABLE(DEPTEMPLOYEES('A00')) AS D
```

# Chapter 37. Calling procedures

Once the activities required to create a procedure (also called a stored procedure) have been completed, a procedure can be invoked by using the CALL statement. The CALL statement is an SQL statement that enables the procedure invocation, the passing of parameters to the procedure, and the receiving of parameters returned from the procedure.

## About this task

Any accessible result sets returned from a procedure can be processed once the procedure has successfully returned. Procedures can be invoked from anywhere that the CALL statement is supported including:

- an embedded SQL client application
- an external routine (procedure, UDF, or method)
- an SQL routine (procedure, UDF, or method)
- an SQL trigger (BEFORE TRIGGER, AFTER TRIGGER, or INSTEAD OF TRIGGER)
- an SQL dynamic compound statement
- from the Command Line Processor (CLP)

If you choose to invoke a procedure from a client application or from an external routine, the client application or external routine can be written in a language other than that of the procedure. For example, a client application written in C++ can use the CALL statement to invoke a procedure written in Java. This provides programmers with great flexibility to program in their language of choice and to integrate code pieces written in different languages.

In addition, the client application that invokes the procedure can be executed on a different platform than the one where the procedure resides. For example a client application running on a Windows operating system can use the CALL statement to invoke a procedure residing on a Linux database server.

An autonomous procedure is a procedure that, when called, executes inside a new transaction independent of the original transaction. When the autonomous procedure successfully completes, it will commit the work performed within the procedure, but if it is unsuccessful, the procedure rolls back any work it performed. Whatever the result of the autonomic procedure, the transaction which called the autonomic procedure is unaffected. To specify a procedure as autonomous, specify the AUTONOMOUS keyword on the CREATE PROCEDURE statement.

When you call a procedure, certain rules apply about exactly which procedure is selected. Procedure selection depends partly on whether you qualify the procedure by specifying the schema name. The DB2 database manager also performs checks based on the number of arguments and any argument names specified in the call to the procedure. See information about the CALL statement for more details about procedure selection.

## Calling procedures from the Command Line Processor (CLP)

You can call stored procedures by using the CALL statement from the DB2 command line processor interface. The stored procedure being called must be defined in the DB2 database system catalog tables.

### Procedure

To call the stored procedure, first connect to the database:

```
db2 connect to sample user userid using password
```

where *userid* and *password* are the user ID and password of the instance where the `sample` database is located.
To use the CALL statement, enter the stored procedure name plus any IN or INOUT parameter values, as well as '?' as a place-holder for each OUT parameter value.
The parameters for a stored procedure are given in the CREATE PROCEDURE statement for the stored procedure in the program source file.

### Example

**SQL procedure examples**

**Example 1.**

In the `whiles.db2` file, the CREATE PROCEDURE statement for the `DEPT_MEDIAN` procedure signature is as follows:

```
CREATE PROCEDURE DEPT_MEDIAN
(IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
```

To invoke this procedure, use the CALL statement in which you must specify the procedure name and appropriate parameter arguments, which in this case are the value for the IN parameter, and a question mark, '?', for the value of the OUT parameter. The procedure's SELECT statement uses the `deptNumber` value on the DEPT column of the STAFF table, so to get meaningful output the IN parameter needs to be a valid value from the DEPT column; for example, the value "51":

```
db2 call dept_median (51, ?)
```

**Note:** On UNIX operating systems, the parentheses have special meaning to the command shell, so they must be preceded with a "\" character or surrounded with quotation marks, as follows:

```
db2 "call dept_median (51, ?)"
```

You do not use quotation marks if you are using the interactive mode of the command line processor.
After running this command, you should receive the following result:

```
Value of output parameters
--------------------------
Parameter Name  : MEDIANSALARY
Parameter Value : +1.76545000000000E+004

Return Status = 0
```

**Example 2.**

This example illustrates how to call a procedure with array parameters. Type phonenumbers is defined as:

```
CREATE TYPE phonenumbers AS VARCHAR(12) ARRAY[1000]
```

Procedure find_customers, defined in the following example, has an IN and an OUT parameter of type phonenumbers. The procedure searches for numbers in numbers_in that begin with the given area_code, and reports them in numbers_out.

```
CREATE PROCEDURE find_customers(
IN numbers_in phonenumbers,
IN area_code CHAR(3),
OUT numbers_out phonenumbers)
BEGIN
DECLARE i, j, max INTEGER;

SET i = 1;
SET j = 1;
SET numbers_out = NULL;
SET max = CARDINALITY(numbers_in);

WHILE i <= max DO
IF substr(numbers_in[i], 1, 3) = area_code THEN
SET numbers_out[j] = numbers_in[i];
SET j = j + 1;
END IF;
SET i = i + 1;
END WHILE;
END
```

To invoke the procedure, you can use the following CALL statement:

```
db2 CALL find_customers(ARRAY['416-305-3745',
                              '905-414-4565',
                              '416-305-3746'],
                              '416',
                               ?)
```

As shown in the CALL statement, when a procedure has an input parameter of an array data type, the input argument can be specified with an array constructor containing a list of literal values.

After running the command, you should receive a result like this:

```
Value of output parameters
  --------------------------
  Parameter Name  : OUT_PHONENUMBERS
  Parameter Value : ['416-305-3745',
                     '416-305-3746']

  Return Status = 0
```

### C stored procedure example

You can also call stored procedures created from supported host languages with the Command Line Processor. In the `samples/c` directory on UNIX, and the `samples\c` directory on Windows, the DB2 database system provides files for creating stored procedures. The `spserver` shared library contains a number of stored procedures that can be created from the source file, `spserver.sqc`. The `spcreate.db2` file catalogs the stored procedures.

In the `spcreate.db2` file, the `CREATE PROCEDURE` statement for the `MAIN_EXAMPLE` procedure begins:

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
                               OUT salary DOUBLE,
                               OUT errorcode INTEGER)
```

To call this stored procedure, you need to put in a CHAR value for the IN parameter, `job`, and a question mark, '?', for each of the OUT parameters.

The procedure's SELECT statement uses the job value on the JOB column of the EMPLOYEE table, so to get meaningful output the IN parameter needs to be a valid value from the JOB column. The C sample program, spclient, that calls the stored procedure, uses 'DESIGNER' for the JOB value. We can do the same, as follows:

```
db2 "call MAIN_EXAMPLE ('DESIGNER', ?, ?)"
```

After running this command, you should receive the following result:

```
Value of output parameters
--------------------------
Parameter Name  : SALARY
Parameter Value : +2.37312500000000E+004

Parameter Name  : ERRORCODE
Parameter Value : 0

Return Status = 0
```

An ERRORCODE of zero indicates a successful result.

# Chapter 38. Working with XML data

## Inserting XML columns

To insert data into an XML column, use the SQL INSERT statement. The input to the XML column must be a well-formed XML document, as defined in the XML 1.0 specification. The application data type can be an XML, character, or binary type.

It is recommended that XML data be inserted from host variables, rather than literals, so that the DB2 database server can use the host variable data type to determine some of the encoding information.

XML data in an application is in its serialized string format. When you insert the data into an XML column, it must be converted to its XML hierarchical format. If the application data type is an XML data type, the DB2 database server performs this operation implicitly. If the application data type is not an XML type, you can invoke the XMLPARSE function explicitly when you perform the insert operation, to convert the data from its serialized string format to the XML hierarchical format.

During document insertion, you might also want to validate the XML document against a registered XML schema. You can do that with the XMLVALIDATE function.

The following examples demonstrate how XML data can be inserted into XML columns. The examples use table MyCustomer, which is a copy of the sample Customer table. The XML data that is to be inserted is in file c6.xml, and looks like this:

```
<customerinfo Cid="1015">
  <name>Christine Haas</name>
  <addr country="Canada">
    <street>12 Topgrove</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N8X-7F8</pcode-zip>
  </addr>
  <phone type="work">905-555-5238</phone>
  <phone type="home">416-555-2934</phone>
</customerinfo>
```

**Example:** In a JDBC application, read XML data from file c6.xml as binary data, and insert the data into an XML column:

```
PreparedStatement insertStmt = null;
String sqls = null;
int cid = 1015;
sqls = "INSERT INTO MyCustomer (Cid, Info) VALUES (?, ?)";
insertStmt = conn.prepareStatement(sqls);
insertStmt.setInt(1, cid);
File file = new File("c6.xml");
insertStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
insertStmt.executeUpdate();
```

**Example:** In a static embedded C application, insert data from a binary XML host variable into an XML column:

```
EXEC SQL BEGIN DECLARE SECTION;
   sqlint64 cid;
   SQL TYPE IS XML AS BLOB (10K) xml_hostvar;
```

```
EXEC SQL END DECLARE SECTION;
...
cid=1015;
/* Read data from file c6.xml into xml_hostvar */
...
EXEC SQL INSERT INTO MyCustomer (Cid,Info) VALUES (:cid, :xml_hostvar);
```

# Querying XML data

You can query or retrieve XML data stored in the database through two main query languages, either by using each language on its own or by using a combination of the two.

The following options are available to you:

- XQuery expressions only
- XQuery expressions that invoke SQL statements
- SQL statements only
- SQL statements that executes XQuery expressions

These various methods allow you to query or retrieve XML and other relational data from either an SQL or XQuery context.

Pieces of or entire XML documents can be queried and retrieved using these methods. Queries can return fragments or entire XML documents, and results returned from queries can be limited by using predicates. Because queries on XML data return XML sequences, a query's result can be used in the construction of XML data as well.

## Comparison of methods for querying XML data

Because XML data can be queried in a number of ways, using XQuery, SQL, or a combination of these, the method to choose can differ depending on your situation. The following sections describe conditions that are advantageous for a particular query method.

### XQuery only

Querying with XQuery alone can be a suitable choice when:

- applications access only XML data, without the need to query non-XML relational data
- migrating queries previously written in XQuery to DB2 for Linux, UNIX, and Windows
- returning query results to be used as values for constructing XML documents
- the query author is more familiar with XQuery than SQL

### XQuery that invokes SQL

Querying with XQuery that invokes SQL can be a suitable choice when (in addition to the scenarios identified in the previous section on using XQuery only):

- queries involve XML data and relational data; SQL predicates and indexes defined on the relational columns can be leveraged in the query
- you want to apply XQuery expressions to the results of:
  - UDF calls, as these cannot be invoked directly from XQuery

– XML values constructed from relational data using SQL/XML publishing functions
– queries that use DB2 Net Search Extender which offers full text search of XML documents but which must be used with SQL

### SQL only

When retrieving XML data using only SQL, without any XQuery, you can query only at the XML column level. For this reason, only entire XML documents can be returned from the query. This usage is suitable when:

- you want to retrieve entire XML documents
- you do not need to query based on values within the stored documents, or where the predicates of your query are on other non-XML columns of the table

### SQL/XML functions that execute XQuery expressions

The SQL/XML functions XMLQUERY and XMLTABLE, as well as the XMLEXISTS predicate, enable XQuery expressions to be executed from within the SQL context. Executing XQuery within SQL can be a suitable choice when:

- existing SQL applications need to be enabled for querying within XML documents. To query within XML documents, XQuery expressions need to be executed, which can be done using SQL/XML
- applications querying XML data need to pass parameter markers to the XQuery expression. (The parameter markers are first bound to XQuery variables in XMLQUERY or XMLTABLE.)
- the query author is more familiar with SQL than XQuery
- both relational and XML data needs to be returned in a single query
- you need to join XML and relational data
- you want to group or aggregate XML data. You can apply the GROUP BY or ORDER BY clauses of a subselect to the XML data (for example, after the XML data has been retrieved and collected in table format by using the XMLTABLE function)

## Indexing XML data

An index over XML data can be used to improve the efficiency of queries on XML documents that are stored in an XML column.

In contrast to traditional relational indexes, where index keys are composed of one or more table columns you specify, an index over XML data uses a particular XML pattern expression to index paths and values in XML documents stored within a single column. The data type of that column must be XML.

Instead of providing access to the beginning of a document, index entries in an index over XML data provide access to nodes within the document by creating index keys based on XML pattern expressions. Because multiple parts of a XML document can satisfy an XML pattern, multiple index keys may be inserted into the index for a single document.

You create an index over XML data using the CREATE INDEX statement, and drop an index over XML data using the DROP INDEX statement. The GENERATE KEY USING XMLPATTERN clause you include with the CREATE INDEX statement specifies what you want to index.

Some of the keywords used with the CREATE INDEX statement for indexes on non-XML columns do not apply to indexes over XML data. The UNIQUE keyword also has a different meaning for indexes over XML data.

## Example: Creating an index over XML data

Suppose that table companyinfo has an XML column named companydocs, which contains XML document fragments like these:

**Document for Company1**

```
<company name="Company1">
  <emp id="31201" salary="60000" gender="Female">
    <name>
      <first>Laura</first>
      <last>Brown</last>
    </name>
    <dept id="M25">
      Finance
    </dept>
  </emp>
</company>
```

**Document for Company2**

```
<company name="Company2">
  <emp id="31664" salary="60000" gender="Male">
    <name>
      <first>Chris</first>
      <last>Murphy</last>
    </name>
    <dept id="M55">
      Marketing
    </dept>
  </emp>
  <emp id="42366" salary="50000" gender="Female">
    <name>
      <first>Nicole</first>
      <last>Murphy</last>
    </name>
    <dept id="K55">
      Sales
    </dept>
  </emp>
</company>
```

Users of the companyinfo table often retrieve employee information using the employee ID. You might use an index like this one to make that retrieval more efficient:

```
CREATE INDEX empindex on companyinfo(companydocs)          1
  GENERATE KEY USING XMLPATTERN '/company/emp/@id'         2
   AS SQL DOUBLE                                     3
```

*Figure 21. Example of an index over XML data*

Notes to Figure 21:

**1** The index over XML data is defined on the companydocs column of the companyinfo table. companydocs must be of the XML data type.

**2** The GENERATE KEY USING XMLPATTERN clause provides information about what you want to index. This clause is called an XML index specification. The XML index specification contains an XML pattern clause. The XML pattern clause in this example indicates that you want to index the values of the id attribute of each employee element.

3 AS SQL DOUBLE indicates that indexed values are stored as DOUBLE values.

## Updating XML data

To update data in an XML column, use the SQL UPDATE statement. Include a WHERE clause when you want to update specific rows.

The entire column value will be replaced. The input to the XML column must be a well-formed XML document. The application data type can be an XML, character, or binary type.

When you update an XML column, you might also want to validate the input XML document against a registered XML schema. You can do that with the XMLVALIDATE function.

You can use XML column values to specify which rows are to be updated. To find values within XML documents, you need to use XQuery expressions. One way of specifying XQuery expressions is the XMLEXISTS predicate, which allows you to specify an XQuery expression and determine if the expression results in an empty sequence. When XMLEXISTS is specified in the WHERE clause, rows will be updated if the XQuery expression returns a non-empty sequence.

The following examples demonstrate how XML data can be updated in XML columns. The examples use table MYCUSTOMER, which is a copy of the sample CUSTOMER table. The examples assume that MYCUSTOMER already contains a row with a customer ID value of 1004. The XML data that updates existing column data is assumed to be stored in a file c7.xml, whose contents look like this:

```
<customerinfo Cid="1004">
  <name>Christine Haas</name>
  <addr country="Canada">
    <street>12 Topgrove</street>
    <city>Toronto</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>N9Y-8G9</pcode-zip>
  </addr>
  <phone type="work">905-555-5238</phone>
  <phone type="home">416-555-2934</phone>
</customerinfo>
```

**Example:** In a JDBC application, read XML data from file c7.xml as binary data, and use it to update the data in an XML column:

```
PreparedStatement updateStmt = null;
String sqls = null;
int cid = 1004;
sqls = "UPDATE MyCustomer SET Info=? WHERE Cid=?";
updateStmt = conn.prepareStatement(sqls);
updateStmt.setInt(1, cid);
File file = new File("c7.xml");
updateStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
updateStmt.executeUpdate();
```

**Example:** In an embedded C application, update data in an XML column from a binary XML host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   sqlint64 cid;
   SQL TYPE IS XML AS BLOB (10K) xml_hostvar;
EXEC SQL END DECLARE SECTION;
...
```

```
cid=1004;
/* Read data from file c7.xml into xml_hostvar */
...
EXEC SQL UPDATE MyCustomer SET Info=:xml_hostvar WHERE Cid=:cid;
```

In these examples, the value of the Cid attribute within the <customerinfo> element happens to be stored in the CID relational column as well. Because of this, the WHERE clause in the UPDATE statements used the relational column CID to specify the rows to update. In the case where the values that determine which rows are chosen for update are found only within the XML documents themselves, the XMLEXISTS predicate can be used. For example, the UPDATE statement in the previous embedded C application example can be changed to use XMLEXISTS as follows:

```
EXEC SQL UPDATE MyCustomer SET Info=:xml_hostvar
       WHERE XMLEXISTS ('$doc/customerinfo[@Cid = $c]'
                        passing INFO as "doc", cast(:cid as integer) as "c");
```

**Example:** The following example updates existing XML data from the MYCUSTOMER table. The SQL UPDATE statement operates on a row of the MYCUSTOMER table and replaces the document in the INFO column of the row with the logical snapshot of the document modified by the transform expression:

```
UPDATE MyCustomer
SET info = XMLQUERY(
   'transform
    copy $newinfo := $info
    modify do insert <status>Current</status>
      as last into $newinfo/customerinfo
    return $newinfo' passing info as "info")
WHERE cid = 1004
```

# Chapter 39. Working with temporal tables and time travel queries

Use SQL statements to store and retrieve time-based data in temporal tables.

## Inserting data into a system-period temporal table

For a user, inserting data into a system-period temporal table is similar to inserting data into a regular table.

### About this task

When inserting data into a system-period temporal table, the database manager automatically generates the values for the row-begin and row-end timestamp columns. The database manager also generates the transaction start-ID value that uniquely identifies the transaction that is inserting the row.

### Procedure

To insert data into a system-period temporal table, use the INSERT statement to add data to the table. For example, the following data was inserted on January 31, 2010 (2010-01-31) to the table created in the example in " Creating a system-period temporal table."

```
INSERT INTO policy_info(policy_id, coverage)
   VALUES('A123',12000);

INSERT INTO policy_info(policy_id, coverage)
   VALUES('B345',18000);

INSERT INTO policy_info(policy_id, coverage)
   VALUES('C567',20000);
```

### Results

The `policy_info` table now contains the following insurance coverage data. The `sys_start`, `sys_end`, and `ts_id` column entries were generated by the database manager.

*Table 25. Data added to a system-period temporal table (policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

The `his_policy_info` history table remains empty because no history rows are generated by an insert.

*Table 26. History table (hist_policy_info) after insert*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| | | | | |

**Note:** The row-begin column, `sys_start`, represents the time when the row data became current. The database manager generates this value by using a reading of the system clock at the moment it executes the first data change statement in the transaction that generates the row. The database manager also generates the transaction start-ID column, `ts_id`, which captures the time when execution started for a transaction that impacts the row. In many cases the timestamp values for both these columns are the same because they result from the execution of the same transaction.

When multiple transactions are updating the same row, timestamp conflicts can occur. The database manager can resolve these conflicts by making adjustments to row-begin column timestamp values. In such cases, the values in row-begin column and transaction start-ID column would differ. The **Example** section in " Updating a system-period temporal table" provides more details on timestamp adjustments.

## Updating data in a system-period temporal table

Updating data in a system-period temporal table results in rows that are added to its associated history table.

### About this task

In addition to updating the values of specified columns in rows of the system-period temporal table, the UPDATE statement inserts a copy of the existing row into the associated history table. The history row is generated as part of the same transaction that updates the row. If a single transactions make multiple updates to the same row, only one history row is generated and that row reflects the state of the record before any changes were made by the transaction.

**Note:** Timestamp value conflicts can occur when multiple transactions are updating the same row. When these conflicts occur, the setting for the "systime_period_adj" database configuration parameter determines whether timestamp adjustments are made or if transactions should fail. The **Multiple changes to a row by different transactions** example in the **More examples** section provides more details. Application programmers might consider using SQLCODE or SQLSTATE values to handle potential timestamp value adjustment-related return codes from SQL statements.

### Procedure

To update data in a system-period temporal table, use the UPDATE statement. For example, it was discovered that were some errors in the insurance coverage levels for a customer and the following data was updated on February 28, 2011 (2011-02-28) in the example table that had data added in the " Inserting data into a system-period temporal table" topic.
The following table contains the original `policy_info` table data.

*Table 27. Original data in the system-period temporal table (policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

- The coverage for policy C567 should be 25000.

```
UPDATE policy_info
   SET coverage = 25000
   WHERE policy_id = 'C567';
```

  The update to policy C567 affects the system-period temporal table and its history table, causing the following things to occur:

  1. The coverage value for the row with policy C567 is updated to 25000.
  2. In the system-period temporal table, the database manager updates the `sys_start` and `ts_id` values to the date of the update.
  3. The original row is moved to the history table. The database manager updates the `sys_end` value to the date of the update. This row can be interpreted as the valid coverage for policy C567 from 2010-01-31-22.31.33.495925000000 to 2011-02-28-09.10.12.649592000000.

*Table 28. Updated data in the system-period temporal table (policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 25000 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 29. History table (hist_policy_info) after update*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| C567 | 20000 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |

## More examples

This section contains more examples of updating system-period temporal tables.

**Time specifications**

In the following example, a time period is specified as part of the table update. The following update is run after the update in the preceding **Procedure** section.

```
UPDATE (SELECT * FROM policy_info
   FOR SYSTEM_TIME AS OF '2010-01-31-22.31.33.495925000000')
   SET coverage = coverage + 1000;
```

This update returns an error because it implicitly attempts to update history rows. The SELECT explicitly queries the `policy_info` table and implicitly queries its associated history table (`hist_policy_info`). The C567 row in `hist_policy_info` would be returned by the SELECT, but rows in a history table that were accessed implicitly cannot be updated.

**Multiple changes to a row by different transactions**

In the following example, two transactions are executing SQL statements against the `policy_info` table at the same time. In this example, the timestamps are simplified to a placeholder instead of a sample system clock value. For example, instead of 2010-01-31-22.31.33.495925000000, the example uses T1. Higher numbered placeholders indicate later actions within the transaction. For example, T5 is later than T4.

When you insert or update multiple rows within a single SQL transaction, the values for the row-begin column are the same for all the impacted rows. That value comes from a reading of the system clock at the moment the first data change statement in the transaction is executed. For example, all times associated with transaction ABC will have a time of T1.

**Transaction ABC**

```
T1: INSERT INTO policy_info
        (policy_id, coverage)
        VALUES ('S777',7000);

T4: UPDATE policy_info
      SET policy_id = 'X999'
      WHERE policy_id = 'T888';

T5: INSERT INTO policy_info
        (policy_id, coverage)
        VALUES ('Y555',9000);

T6: COMMIT;
```

**Transaction XYZ**

```
T2: INSERT INTO policy_info
        (policy_id, coverage)
        VALUES ('T888',8000);

T3: COMMIT;
```

After the inserts at **T1** and **T2**, the `policy_info` table would look like this and the history table would be empty (`hist_policy_info`). The value **max** in the `sys_end` column is populated with the maximum default value for the TIMESTAMP(12) data type.

*Table 30. Different transaction inserts to the policy_info table*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| S777 | 7000 | T1 | max | T1 |
| T888 | 8000 | T2 | max | T2 |

After the update by transaction ABC at time T4, the policy information looks like the following tables. All the rows in the `policy_info` table reflect the insert and update activities from transaction ABC. The `sys_start` and `ts_id` columns for these rows are populated with time T1, which is the time of the first data change statement in transaction ABC. The policy information inserted by transaction XYZ was updated and the original row is moved to the history table.

*Table 31. Different transactions after update to the policy_info table*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| S777 | 7000 | T1 | max | T1 |
| X999 | 8000 | T1 | max | T1 |

*Table 32. History table after different transactions update (hist_policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| T888 | 8000 | T2 | T1 | T2 |

The history table shows a `sys_end` time that is less than the `sys_start`. In this situation, the update at time T4 could not execute and transaction ABC would fail (SQLSTATE 57062, SQLCODE SQL20528N). To avoid such failures, the **systime_period_adj** database configuration parameter can be set to *YES* which allows the database manager to adjust the row-begin timestamp (SQLSTATE 01695, SQLCODE SQL5191W). The `sys_start` timestamp for the time T4 update in transaction ABC is set to time T2 plus a delta (T2+delta). This adjustment only applies to the time T4 update, all other changes made by transaction ABC would continue to use the time T1 timestamp (for example, the insert of the policy with `policy_id` Y555). After this adjustment and the completion of transaction ABC, the insurance policy tables would contain the following data:

*Table 33. Different transactions after time adjustment (policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| S777 | 7000 | T1 | max | T1 |
| X999 | 8000 | T2+delta | max | T1 |
| Y555 | 9000 | T1 | max | T1 |

*Table 34. History table after time adjustment (hist_policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| T888 | 8000 | T2 | T2+delta | T2 |

**Multiple changes to a row in the same transaction**

In the following example, a transaction makes multiple changes to a row. Using the insurance policy tables from the previous example, transaction ABC continues and updates the policy with `policy_id` X999 at time T6 (originally T6 was a COMMIT statement).

**Transaction ABC**

```
T6: UPDATE policy_info SET policy_id = 'R111' WHERE policy_id = 'X999';
T7: COMMIT;
```

This row has now experienced the following changes:

1. Created as policy T888 by transaction XYZ at time T2.
2. Updated to policy X999 by transaction ABC at time T4.
3. Updated to policy R111 by transaction ABC at time T6.

When a transaction makes multiple updates to the same row, the database manager generates a history row only for the first change. This, results in the following tables:

*Table 35. Same transaction after updates (policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| S777 | 7000 | T1 | max | T1 |
| R111 | 8000 | T1 | max | T1 |
| Y555 | 9000 | T1 | max | T1 |

*Table 36. History table after same transaction update (hist_policy_info)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| T888 | 8000 | T2 | T2+delta | T2 |

The database manager uses the transaction-start-ID (`ts_id`) to uniquely identify the transaction that changes the row. When multiple rows are inserted or updated within a single SQL transaction, then the values for the transaction start-ID column are the same for all the rows and are unique from the values generated for this column by other transactions. Before generating a history row, the database manager determines that the last update to the row was for the transaction that started at time T1 (`ts_id` is T1), which is the same transaction start time for the transaction that makes the current change and so no history row is generated. The `sys_start` value for the row in the `policy_info` table is changed to time T1.

**Updating a view**

A view that references a system-period temporal table or a bitemporal table can be updated only if the view definition does not contain a FOR SYSTEM_TIME clause. The following UPDATE statement updates the `policy_info` table and generates history rows.

```
CREATE VIEW viewA AS SELECT * FROM policy_info;
UPDATE viewA SET col2 = col2 + 1000;
```

A view that references a system-period temporal table or a bitemporal table with a view definition containing a FOR SYSTEM_TIME clause can be made updatable by defining an INSTEAD OF trigger. The following example updates the `regular_table` table.

```
CREATE VIEW viewB AS SELECT * FROM policy_info;
   FOR SYSTEM_TIME BETWEEN
   TIMESTAMP '2010-01-01 10:00:00' AND TIMESTAMP '2011-01-01 10:00:00';

CREATE TRIGGER update INSTEAD OF UPDATE ON viewB
   REFERENCING NEW AS n FOR EACH ROW
   UPDATE regular_table SET col1 = n.id;

UPDATE viewB set id = 500;
```

# Deleting data from a system-period temporal table

Deleting data from a system-period temporal table removes rows from the table and adds rows to the associated history table. The rows are added with the appropriate system timestamps.

## About this task

In addition to deleting the specified rows of the system-period temporal table, the DELETE FROM statement moves a copy of the existing row into the associated history table before the row is deleted from the system-period temporal table.

## Procedure

To delete data from a system-period temporal table, use the DELETE FROM statement. For example, the owner of policy B345 decides to cancel insurance coverage. The data was deleted on September 1, 2011 (2011-09-01) from the table that was updated in the " Updating data in a bitemporal table" topic.

```
DELETE FROM policy_info WHERE policy_id = 'B345';
```

## Results

The original `policy_info` table data is as follows:

*Table 37. Data in the system-period temporal table (policy_info) before the DELETE statement*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 25000 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

The deletion of policy B345 affects the system-period temporal table and its history table, causing the following things to occur:

1. The row where the `policy_id` column value is B345 is deleted from the system-period temporal table.
2. The original row is moved to the history table. The database manager updates the `sys_end` column value to the date of the delete.

*Table 38. Data in the system-period temporal table (policy_info) after the DELETE statement*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 25000 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 39. History table (hist_policy_info) after delete*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| C567 | 20000 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 2011-09-01-12.18.22.959254000000 | 2010-01-31-22.31.33.495925000000 |

## Example

This section contains more examples of delete operations on system-period temporal tables.

**Time specifications**

In the following example, a time period is specified as part of the DELETE statement. The following delete is run after the delete in the preceding **Procedure** section.

```
DELETE FROM (SELECT * FROM policy_info
   FOR SYSTEM_TIME AS OF '2010-01-31-22.31.33.495925000000')
   WHERE policy_id = C567;
```

This DELETE statement returns an error. The SELECT statement explicitly queries the `policy_info` table and implicitly queries its associated history table (`hist_policy_info`). The row with a `policy_id` column value of C567 in the `hist_policy_info` table would be returned by the SELECT statement, but rows in a history table that were accessed implicitly cannot be deleted.

## Querying system-period temporal data

Querying a system-period temporal table can return results for a specified point or period in time. Those results can include current values and previous historic values.

### About this task

When querying a system-period temporal table, you can include FOR SYSTEM_TIME in the FROM clause. Using FOR SYSTEM_TIME specifications, you can query the current and past state of your data. Time periods are specified as follows:

**AS OF** *value1*
: Includes all the rows where the begin value for the period is less than or equal to *value1* and the end value for the period is greater than *value1*. This enables you to query your data as of a certain point in time.

**FROM** *value1* **TO** *value2*
: Includes all the rows where the begin value for the period is equal to or greater than *value1* and the end value for the period is less than *value2*. This means that the begin time is included in the period, but the end time is not.

**BETWEEN** *value1* **AND** *value2*
: Includes all the rows where any time period overlaps any point in time between *value1* and *value2*. A row is returned if the begin value for the period is less than or equal to *value2* and the end value for the period is greater than *value1*.

See the following section for some sample queries.

### Procedure

To query a system-period temporal table, use the SELECT statement. For example, each of the following queries requests policy information from the result tables in the Deleting data from a system-period temporal table topic. Each query uses a variation of the FOR SYSTEM_TIME specification.
The `policy_info` table and its associated history table are as follows:

*Table 40. System-period temporal table: policy_info*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

*Table 40. System-period temporal table: policy_info (continued)*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| C567 | 25000 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 41. History table: hist_policy_info*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| C567 | 20000 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 2011-09-01-12.18.22.959254000000 | 2010-01-31-22.31.33.495925000000 |

- Query with no time period specification. For example:

```
SELECT policy_id, coverage
   FROM policy_info
   where policy_id = 'C567'
```

  This query returns one row. The SELECT queries only the `policy_info` table. The history table is not queried because FOR SYSTEM_TIME was not specified.

```
C567, 25000
```

- Query with FOR SYSTEM_TIME AS OF specified. For example:

```
SELECT policy_id, coverage
   FROM policy_info
   FOR SYSTEM_TIME AS OF
      '2011-02-28-09.10.12.649592000000'
```

  This query returns three rows. The SELECT queries both the `policy_info` and the `hist_policy_info` tables. The begin column of a period is inclusive, while the end column is exclusive. The history table row with a sys_end column value of 2011-02-28-22.31.33.495925000000 equals *value1,* but it must be less than *value1* in order to be returned.

```
A123, 12000
C567, 25000
B345, 18000
```

- Query with FOR SYSTEM_TIME FROM..TO specified. For example:

```
SELECT policy_id, coverage, sys_start, sys_end
   FROM policy_info
   FOR SYSTEM_TIME FROM
      '0001-01-01-00.00.00.000000' TO '9999-12-30-00.00.00.000000000000'
   where policy_id = 'C567'
```

  This query returns two rows. The SELECT queries both the `policy_info` and the `hist_policy_info` tables.

```
C567, 25000, 2011-02-28-09.10.12.649592000000, 9999-12-30-00.00.00.000000000000
C567, 20000, 2010-01-31-22.31.33.495925000000, 2011-02-28-09.10.12.649592000000
```

- Query with FOR SYSTEM_TIME BETWEEN..AND specified. For example:

```
SELECT policy_id, coverage
   FROM policy_info
   FOR SYSTEM_TIME BETWEEN
      '2011-02-28-09.10.12.649592000000' AND '9999-12-30-00.00.00.000000000000'
```

This query returns three rows. The SELECT queries both the policy_info and the hist_policy_info tables. The rows with a sys_start column value of 2011-02-28-09.10.12.649592000000 are equal to *value1* and are returned because the begin time of a period is included. The rows with a sys_end column value of 2011-02-28-09.10.12.649592000000 are equal to *value1* and are not returned because the end time of a period is not included.

```
A123, 12000
C567, 25000
B345, 18000
```

### More examples

This section contains more querying system-period temporal table examples.

**Query using other valid date or timestamp values**

The policy_info table was created with its time-related columns declared as TIMESTAMP(12), so queries using any other valid date or timestamp value are converted to use TIMESTAMP(12) before execution. For example:

```
SELECT policy_id, coverage
   FROM policy_info
   FOR SYSTEM_TIME AS OF '2011-02-28'
```

is converted and executed as:

```
SELECT policy_id, coverage
   FROM policy_info
   FOR SYSTEM_TIME AS OF '2011-02-28-00.00.00.000000000000'
```

**Querying a view**

A view can be queried as if it were a system-period temporal table. FOR SYSTEM_TIME specifications can be specified after the view reference.

```
CREATE VIEW policy_2011(policy, start_date)
   AS SELECT policy_id, sys_start FROM policy_info;

SELECT * FROM policy_2011 FOR SYSTEM_TIME BETWEEN
   '2011-01-01-00.00.00.000000' AND '2011-12-31-23.59.59.999999999999';
```

The SELECT on the view policy_2011 queries both the policy_info and the hist_policy_info tables. Returned are all policies that were active at anytime in 2011 and includes the date the policies were started.

```
A123, 2010-01-31-22.31.33.495925000000
C567, 2011-02-28-09.10.12.649592000000
C567, 2010-01-31-22.31.33.495925000000
B345, 2010-01-31-22.31.33.495925000000
```

If a view definition contains a period specification, then queries against the view cannot contain period specifications. The following statements return an error due to multiple period specifications:

```
CREATE VIEW all_policies AS SELECT * FROM policy_info;
   FOR SYSTEM_TIME AS OF '2011-02-28-09.10.12.649592000000';

SELECT * FROM all_policies FOR SYSTEM_TIME BETWEEN
   '2011-01-01-00.00.00.000000' AND '2011-12-31-23.59.59.999999999999';
```

## Setting the system time for a session

Setting the system time with the CURRENT TEMPORAL SYSTEM_TIME special register can reduce or eliminate the changes required when running an application against different points in time.

## About this task

When you have an application that you want to run against a system-period temporal table to query the state of your business for a number of different dates, you can set the date in a special register. If you need to query your data as of today, as of the end of the last quarter, and as of the same date from last year, it might not be possible to change the application and add AS OF specifications to each SQL statement. This restriction is likely the case when you are using packaged applications. To address such scenarios, you can use the CURRENT TEMPORAL SYSTEM_TIME special register to set the date or timestamp at the session level.

Setting the CURRENT TEMPORAL SYSTEM_TIME special register does not affect regular tables. Only queries on temporal tables with versioning enabled (system-period temporal tables and bitemporal tables) use the time set in the special register. There is also no affect on DDL statements. The special register does not apply to any scans run for referential integrity processing. .

**Important:** When the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, data modification statements like INSERT, UPDATE, and DELETE against system-period temporal tables are blocked. If the special register was set to some time in the past, for example five years ago, then allowing data modification operations might result in changes to your historical data records. Utilities like IMPORT and LOAD are also blocked against system-period temporal tables when the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value.

The BIND command contains the SYSTIMESENSITIVE option that indicates whether references to system-period temporal tables in static and dynamic SQL statements are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. For SQL procedures, use the SET_ROUTINE_OPTS procedure to set the bind-like options, called query compiler variables.

## Procedure

When this special register is set to a non-null value, applications that issue a query will return data as of that date or timestamp. The following examples request information from the result tables in the Deleting data from a system-period temporal table topic.

- Set the special register to the current timestamp and query data from one year ago. Assuming a current timestamp of 2011-05-17-14.45.31.434235000000:

```
SET CURRENT TEMPORAL SYSTEM_TIME = CURRENT TIMESTAMP - 1 YEAR;
SELECT policy_id, coverage FROM policy_info;
```

- Set the special register to a timestamp and reference a system-period temporal table in view definitions.

```
CREATE VIEW view1 AS SELECT policy_id, coverage FROM policy_info;
CREATE VIEW view2 AS SELECT * FROM regular_table
   WHERE col1 IN (SELECT coverage FROM policy_info);
SET CURRENT TEMPORAL SYSTEM_TIME = TIMESTAMP '2011-02-28-09.10.12.649592000000';
SELECT * FROM view1;
SELECT * FROM view2;
```

- Set the special register to the current timestamp and issue a query that contains a time period specification. Assuming a current timestamp of 2011-05-17-14.45.31.434235000000:

```
SET CURRENT TEMPORAL SYSTEM_TIME = CURRENT TIMESTAMP - 1 YEAR;
SELECT *
  FROM policy_info FOR SYSTEM_TIME AS OF '2011-02-28-09.10.12.649592000000';
```

## Results

The policy_info table and its associated history table are as follows:

*Table 42. Data in the system-period temporal table (policy_info) after the DELETE statement*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| A123 | 12000 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 25000 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 43. History table (hist_policy_info) after delete*

| policy_id | coverage | sys_start | sys_end | ts_id |
|---|---|---|---|---|
| C567 | 20000 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2010-01-31-22.31.33.495925000000 | 2011-09-01-12.18.22.959254000000 | 2010-01-31-22.31.33.495925000000 |

- The request for data from one year ago queries the policy_info table as of 2010-05-17-14.45.31.434235000000. The query is implicitly rewritten to:

  ```
  SELECT policy_id, coverage FROM policy_info
     FOR SYSTEM_TIME AS OF TIMESTAMP '2010-05-17-14.45.31.434235000000';
  ```

  The SELECT queries both the policy_info and the hist_policy_info tables and returns:

  ```
  A123, 12000
  C567, 20000
  B345, 18000
  ```

- The query on view1 is implicitly rewritten to:

  ```
  SELECT * FROM view1 FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME;
  ```

  The query is then rewritten to:

  ```
  SELECT policy_id, coverage FROM policy_info
     FOR SYSTEM_TIME AS OF TIMESTAMP '2011-02-28-09.10.12.649592000000';
  ```

  The SELECT queries both the policy_info and the hist_policy_info tables and returns:

  ```
  A123, 12000
  C567, 25000
  B345, 18000
  ```

  The query on view2 involves a view on a regular table that references a system-period temporal table, causing an implicit relationship between a regular table and the special register. The query is implicitly rewritten to:

  ```
  SELECT * FROM view2 FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME;
  ```

  The query is then rewritten to:

  ```
  SELECT * FROM regular_table WHERE col1 in (SELECT coverage FROM policy_info
     FOR SYSTEM_TIME AS OF TIMESTAMP '2011-02-28-09.10.12.649592000000');
  ```

The SELECT returns rows where col1 values match values in coverage.

- An error is returned because there are multiple time period specifications. The special register was set to a non-null value and the query also specified a time.

## Inserting data into an application-period temporal table

Inserting data into an application-period temporal table is similar to inserting data into a regular table.

### About this task

When inserting data into an application-period temporal table, the only special consideration is the need to include the row-begin and row-end columns that capture when the row is valid from the perspective of the associated business applications. This valid period is called the BUSINESS_TIME period. The database manager automatically generates an implicit check constraint that ensures that the begin column of the BUSINESS_TIME period is less than its end column. If a unique constraint or index with BUSINESS_TIME WITHOUT OVERLAPS was created for the table, you must ensure that no BUSINESS_TIME periods overlap.

### Procedure

To insert data into an application-period temporal table, use the INSERT statement to add data to the table. For example, the following data was inserted to the table created in the example in Creating an application-period temporal table topic.

```
INSERT INTO policy_info VALUES('A123',12000,'2008-01-01','2008-07-01');

INSERT INTO policy_info VALUES('A123',16000,'2008-07-01','2009-01-01');

INSERT INTO policy_info VALUES('A123',16000,'2008-06-01','2008-08-01');

INSERT INTO policy_info VALUES('B345',18000,'2008-01-01','2009-01-01');

INSERT INTO policy_info VALUES('C567',20000,'2008-01-01','2009-01-01');
```

### Results

There were five INSERT statements issued, but only four rows were added to the table. The second and third INSERT statements are attempting to add rows for policy A123, but their BUSINESS_TIME periods overlap which results in the following:

- The second insert adds a row for policy_id A123 with a bus_start value of 2008-07-01 and a bus_end value of 2009-01-01.
- The third insert attempts to add a row for policy_id A123, but it fails because its BUSINESS_TIME period overlaps that of the previous insert. The policy_info table was created with a BUSINESS_TIME WITHOUT OVERLAPS index and the third insert has a bus_end value of 2008-08-01, which is within the time period of the earlier insert.

The begin column of a period is inclusive, while the end column is exclusive, meaning that the row with a bus_end value of 2008-07-01 does not have a BUSINESS_TIME period overlap with the row that contains a bus_start value of 2008-07-01. As a result, the policy_info table now contains the following insurance coverage data:

*Table 44. Data added to an application-period temporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 |

# Updating data in an application-period temporal table

Updating data in an application-period temporal table can be similar to updating data in a regular table, but data can also be updated for specified points of time in the past, present, or future. Point in time updates can result in rows being split and new rows being inserted automatically into the table.

## About this task

In addition to the regular UPDATE statement, application-period temporal tables also support time range updates where the UPDATE statement includes the FOR PORTION OF BUSINESS_TIME clause. A row is a candidate for updating if its period-begin column, period-end column, or both fall within the range specified in the FOR PORTION OF BUSINESS_TIME clause.

## Procedure

To update data in an application-period temporal table, use the UPDATE statement. For example, you discovered some errors in the insurance coverage information for some customers and the following updates are performed on the sample table that was introduced in the " Inserting data into an application-period temporal table" topic.
The following table contains the original `policy_info` table data.

*Table 45. Original data in the application-period temporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 |

The `policy_info` table was created with a BUSINESS_TIME WITHOUT OVERLAPS index. When using the regular UPDATE statement, you must ensure that no BUSINESS_TIME periods overlap. Updating an application-period temporal table by using the FOR PORTION OF BUSINESS_TIME clause avoids period overlap problems. This clause causes rows to be changed and can result in rows that are inserted when the existing time period for a row that is being updated is not fully contained within the range specified in the UPDATE statement.

- The coverage for policy B345 actually started on March 1, 2008 (2008-03-01) and the coverage should be 18500:

```
UPDATE policy_info
  SET coverage = 18500, bus_start = '2008-03-01'
  WHERE policy_id = 'B345'
  AND coverage=18000
```

The update to policy B345 uses a regular UPDATE statement. There is only one row in the policy_info table for policy_id B345, so there are no potential BUSINESS_TIME periods overlaps. As a result, the bus_start column value is updated to 2008-03-01 and the coverage value is updated to 18500. Note that updates to a BUSINESS_TIME period column cannot include the FOR PORTION OF BUSINESS_TIME clause.

*Table 46. Policy B345 updated*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 |
| B345 | 18500 | 2008-03-01 | 2009-01-01 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 |

- The coverage for policy C567 should be 25000 for the year 2008:

```
UPDATE policy_info
   FOR PORTION OF BUSINESS_TIME FROM '2008-01-01' TO '2009-01-01'
   SET coverage = 25000
   WHERE policy_id = 'C567';
```

The update to policy C567 applies to the BUSINESS_TIME period from 2008-01-01 to 2009-01-01. There is only one row in the policy_info table for policy_id C567 that includes this time period. The BUSINESS_TIME period is fully contained within the bus_start and bus_end column values for that row. As a result, the coverage value is updated to 25000. The bus_start and bus_end column values are unchanged.

*Table 47. Policy C567 updated*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 |
| B345 | 18500 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 |

- The coverage for policy A123 shows an increase from 12000 to 16000 on July 1 (2008-07-01), but an earlier increase to 14000 is missing:

```
UPDATE policy_info
   FOR PORTION OF BUSINESS_TIME FROM '2008-06-01' TO '2008-08-01'
   SET coverage = 14000
   WHERE policy_id = 'A123';
```

The update to policy A123 applies to the BUSINESS_TIME period from 2008-06-01 to 2008-08-01. There are two rows in the policy_info table for policy_id A123 that include part of this time period.

1. The BUSINESS_TIME period is partially contained in the row that has a bus_start value of 2008-01-01 and a bus_end value of 2008-07-01. This row overlaps the beginning of the specified period because the earliest time value in the BUSINESS_TIME period is greater than the rows bus_start value, but less than its bus_end value.
2. The BUSINESS_TIME period is partially contained in the row that has a bus_start value of 2008-07-01 and a bus_end value of 2009-01-01. This row

overlaps the end of the specified period because the latest time value in the BUSINESS_TIME period is greater than the rows bus_start value, but less than its bus_end value.

As a result, the update causes the following things to occur:

1. When the bus_end value overlaps the beginning of the specified period, the row is updated to the new coverage value of 14000. In this updated row, the bus_start value is set to 2008-06-01 which is the begin value of the UPDATE specified period, and the bus_end value is unchanged. An additional row is inserted with the original values from the row, except that the bus_end value is set to 2008-06-01. This new row reflects the BUSINESS_TIME period when coverage was 12000.
2. When the bus_start value overlaps the end of the specified period, the row is updated to the new coverage value of 14000. In this updated row, the bus_start value is unchanged and the bus_end value is set to 2008-08-01 which is the end value of the UPDATE specified period. An additional row is inserted with the original values from the row, except that the bus_start value is set to 2008-08-01. This new row reflects the BUSINESS_TIME period when coverage was 16000.

*Table 48. Policy A123 updated*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 |
| B345 | 18500 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 |

## More examples

This section contains more updating application-period temporal table examples.

**Merging content**

In the following example, a MERGE statement uses the FOR PORTION OF clause to update the policy_info table with the contents of another table (merge_policy).

*Table 49. Content of the merge_policy table*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| C567 | 30000 | 2008-10-01 | 2010-05-01 |
| H789 | 16000 | 2008-10-01 | 2010-05-01 |

1. Create global variables to hold the FROM and TO dates for the FOR PORTION OF clause.

```
CREATE VARIABLE sdate DATE default '2008-10-01';
CREATE VARIABLE edate DATE default '2010-05-01';
```

2. Issue a MERGE statement that merges the content of merge_policy into the policy_info table that resulted from the updates in the preceding "Procedure" section.

```
MERGE INTO policy_info pi1
   USING (SELECT policy_id, coverage, bus_start, bus_end
            FROM merge_policy) mp2
```

```
   ON (pi1.policy_id = mp2.policy_id)
WHEN MATCHED THEN
     UPDATE FOR PORTION OF BUSINESS_TIME FROM sdate TO edate
        SET pi1_coverage = mp2.coverage
WHEN NOT MATCHED THEN
      INSERT (policy_id, coverage, bus_start, bus_end)
       VALUES (mp2.policy_id, mp2.coverage, mp2.bus_start, mp2.bus_end)
```

The policy_id C567 is common to both tables. The C567 bus_start value in merge_policy overlaps the C567 bus_end value in policy_info. This statement results in the following items:

- The bus_end value for coverage of 25000 is set to 2008-10-01.
- A new row is inserted for coverage of 30000 with the bus_start and bus_end values from merge_policy.

The policy_id H789 exists only in merge_policy and so a new row is added to policy_info.

*Table 50. Merged updated data in an application-period temporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2008-10-01 |
| C567 | 30000 | 2008-10-01 | 2010-05-01 |
| H789 | 16000 | 2008-10-01 | 2010-05-01 |

**Update targets**

The FOR PORTION OF BUSINESS_TIME clause can be used only when the target of the update statement is a table or a view. The following updates return errors.

```
UPDATE (SELECT * FROM policy_info) FOR PORTION OF BUSINESS_TIME
   FROM '2008-01-01' TO '06-15-2008' SET policy_id = policy_id + 1;

UPDATE (SELECT * FROM policy_info FOR BUSINESS_TIME AS OF '2008-01-01')
   FOR PORTION OF BUSINESS_TIME FROM '2008-01-01' TO '06-15-2008'
   SET policy_id = policy_id + 1;
```

**Updating a view**

A view with references to an application-period temporal table is updatable. The following UPDATE would update the policy_info table.

```
CREATE VIEW viewC AS SELECT * FROM policy_info;
UPDATE viewC SET coverage = coverage + 5000;
```

A view with an application-period temporal table in its FROM clause that contains a period specification is also updatable. This condition differs from views on system-period temporal tables and bitemporal tables.

```
CREATE VIEW viewD AS SELECT * FROM policy_info
   FOR BUSINESS_TIME AS OF CURRENT DATE;
UPDATE viewD SET coverage = coverage - 1000;
```

A FOR PORTION OF update clause can be included against views with references to application-period temporal tables or bitemporal tables. Such updates are propagated to the temporal tables referenced in the FROM clause of the view definition.

```
CREATE VIEW viewE AS SELECT * FROM policy_info;
UPDATE viewE FOR PORTION OF BUSINESS_TIME
   FROM '2009-01-01' TO '2009-06-01' SET coverage = coverage + 500;
```

# Deleting data from an application-period temporal table

Deleting data from an application-period temporal table removes rows from the table and can potentially result in new rows that are inserted into the application-period temporal table itself.

## About this task

In addition to the regular DELETE statement, application-period temporal tables also support time range deletes where the DELETE statement includes the FOR PORTION OF BUSINESS_TIME clause. A row is a candidate for deletion if its period-begin column, period-end column, or both fall within the range specified in the FOR PORTION OF BUSINESS_TIME clause.

## Procedure

To delete data from an application-period temporal table, use the DELETE FROM statement to delete data. For example, it was discovered that policy A123 should not provide coverage from June 15, 2008 to August 15, 2008 and therefore that data should be deleted from the table that was updated in the Updating data in an application-period temporal table topic.

```
DELETE FROM policy_info
   FOR PORTION OF BUSINESS_TIME FROM '2008-06-15' TO '2008-08-15'
   WHERE policy_id = 'A123';
```

## Results

The original `policy_info` table data is as follows:

*Table 51. Data in the application-period temporal table (policy_info) before the DELETE statement*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 |

Deleting data from an application-period temporal table by using the FOR PORTION OF BUSINESS_TIME clause causes rows to be deleted and can result in rows that are inserted when the time period for a row covers a portion of the range specified in the DELETE FROM statement. Deleting data related to policy

A123 applies to the BUSINESS_TIME period from 2008-06-15 to 2008-08-15. There are three rows in the `policy_info` table for `policy_id` A123 that include all or part of that time period.

The update to policy A123 affects the system-period temporal table and its history table, causing the following the things to occur:

- There is one row where the BUSINESS_TIME period in the DELETE FROM statement covers the entire time period for a row. The row with a `bus_start` value of 2008-07-01 and a `bus_end` value of 2008-08-01 is deleted.
- When only the `bus_end` value falls into the specified period, the row is deleted. A new row is inserted with the original values from the deleted row, except that the `bus_end` value is set to 2008-06-15.
- When only the `bus_start` value falls into the specified period, the row is deleted. A new row is inserted with the original values from the deleted row, except that the `bus_start` value is set to 2008-08-15.

*Table 52. Data in the application-period temporal table (policy_info) after the DELETE statement*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 |
| A123 | 14000 | 2008-06-01 | 2008-06-15 |
| A123 | 16000 | 2008-08-15 | 2009-01-01 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 |

### Example

This section contains more deleting application-period temporal table examples.

**Delete targets**

The FOR PORTION OF BUSINESS_TIME clause can be used only when the target of the delete statement is a table or a view. The following DELETE statement returns an error:

```
DELETE FROM (SELECT * FROM policy_info) FOR PORTION OF BUSINESS_TIME
   FROM '2008-01-01' TO '2008-06-15';
```

## Querying application-period temporal data

Querying an application-period temporal table can return results for a specified time period.

### About this task

When querying an application-period temporal table, you can include FOR BUSINESS_TIME in the FROM clause. Using FOR BUSINESS_TIME specifications, you can query the current, past, and future state of your data. Time periods are specified as follows:

**AS OF** *value1*

Includes all the rows where the begin value for the period is less than or equal to *value1* and the end value for the period is greater than *value1*.

**FROM** *value1* **TO** *value2*

Includes all the rows where the begin value for the period is greater than

or equal to *value1* and the end value for the period is less than *value2*. This means that the begin time is included in the period, but the end time is not.

**BETWEEN** *value1* **AND** *value2*
Includes all the rows where any time period overlaps any point in time between *value1* and *value2*. A row is returned if the begin value for the period is less than or equal to *value2* and the end value for the period is greater than *value1*.

See the following section for some sample queries.

## Procedure

To query an application-period temporal table, use the SELECT statement. For example, each of the following queries requests policy information for `policy_id` A123 from the result table in the " Updating data in an application-period temporal table" topic. Each query uses a variation of the time period specification. The `policy_info` table is as follows:

*Table 53. Application-period temporal table: policy_info*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 |
| A123 | 14000 | 2008-06-01 | 2008-06-15 |
| A123 | 16000 | 2008-08-15 | 2009-01-01 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 |

- Query with no time period specification. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   where policy_id = 'A123'
```

  This query returns all three rows for policy A123.

```
A123, 12000, 2008-01-01, 2008-06-01
A123, 14000, 2008-06-01, 2008-06-15
A123, 16000, 2008-08-15, 2009-01-01
```

- Query with FOR BUSINESS_TIME AS OF specified. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   FOR BUSINESS_TIME AS OF '2008-07-15'
   where policy_id = 'A123'
```

  This query does not return any rows. There are no rows for A123 where the begin value for the period is less than or equal to *2008-07-15* and the end value for the period is greater than *2008-07-15*. Policy A123 had no coverage on *2008-07-15*.

- Query with FOR BUSINESS_TIME FROM...TO specified. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   FOR BUSINESS_TIME FROM
      '2008-01-01' TO '2008-06-15'
   where policy_id = 'A123'
```

This query returns two rows. The begin-column of a period is inclusive, while the end-column is exclusive. The row with a bus_end value of 2008-06-15 is valid until 06-14-2008 at midnight and so is less than *value2*.

```
A123, 12000, 2008-01-01, 2008-06-01
A123, 14000, 2008-06-01, 2008-06-15
```

- Query with FOR BUSINESS_TIME BETWEEN...AND specified. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   FOR BUSINESS_TIME BETWEEN
      '0001-01-01' AND '2008-01-01'
```

This query returns two rows. The rows with a bus_start value of 2008-01-01 are equal to *value1* and are returned because the begin time of a period is included. Note that if a row had a bus_end column value of 2008-01-01, that row would be returned because its end time is equal to *value1* and the end time of a period is included.

```
A123, 12000, 2008-01-01, 2008-06-01
C567, 25000, 2008-01-01, 2009-01-01
```

### More examples

This section contains more querying application-period temporal table examples.

**Querying a view**

A view can be queried as if it were an application-period temporal table. Time period specifications (FOR BUSINESS_TIME) can be specified after the view reference.

```
CREATE VIEW policy_year_end(policy, amount, start_date, end_date)
   AS SELECT * FROM policy_info;

SELECT * FROM policy_year_end FOR BUSINESS_TIME AS OF '2008-12-31';
```

The SELECT on the view `policy_year_end` queries the `policy_info` table and returns all policies that were in effect at the end of 2008.

```
A123, 16000, 2008-08-15, 2009-01-01
B345, 18000, 2008-03-01, 2009-01-01
C567, 25000, 2008-01-01, 2009-01-01
```

If a view definition contains a period specification, then queries against the view cannot contain period specifications. The following statements return an error due to multiple period specifications:

```
CREATE VIEW all_policies AS SELECT * FROM policy_info;
   FOR BUSINESS_TIME AS OF '2008-02-28';

SELECT * FROM all_policies FOR BUSINESS_TIME BETWEEN
   FOR BUSINESS_TIME AS OF '2008-10-01';
```

## Setting the application time for a session

Setting the application time in the CURRENT TEMPORAL BUSINESS_TIME special register can reduce or eliminate the changes required when running an application against different points in time.

### About this task

When you have an application that you want to run against an application-period temporal table to query the state of your business for a number of different dates,

you can set the date in a special register. If you need to query your data AS OF today, AS OF the end of the last quarter, or if you are simulating future events, AS OF some future date, it might not be possible to change the application and add AS OF specifications to each SQL statement. This restriction is likely the case when you are using packaged applications. To address such scenarios, you can use the CURRENT TEMPORAL BUSINESS_TIME special register to set the date at the session level.

Setting the CURRENT TEMPORAL BUSINESS_TIME special register does not affect regular tables. Only queries on temporal tables with a BUSINESS_TIME period enabled (application-period temporal tables and bitemporal tables) use the time set in the special register. There is also no affect on DDL statements.

**Note:** When the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value, data modification statements like INSERT, UPDATE, DELETE, and MERGE against application-period temporal tables are supported. This behavior differs from the CURRENT TEMPORAL SYSTEM_TIME special register which blocks data modification statements against system-period temporal table and bitemporal tables.

The setting for the BUSTIMESENSITIVE bind option determines whether references to application-period temporal tables and bitemporal tables in both static SQL statements and dynamic SQL statements in a package are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. The bind option can be set to YES or NO. For SQL procedures, use the SET_ROUTINE_OPTS procedure to set the bind-like options, called query compiler variables.

## Procedure

When this special register is set to a non-null value, applications that issue a query returns data as of that date. The following examples request information from the result tables in the " Deleting data from an application-period temporal table "topic.

- Set the special register to a non-null value and query data as of that date. For example:

```
SET CURRENT TEMPORAL BUSINESS_TIME = '2008-01-01';
SELECT * FROM policy_info;
```

- Set the special register to a time and reference an application-period temporal table in view definitions.

```
CREATE VIEW view1 AS SELECT policy_id, coverage FROM policy_info;
CREATE VIEW view2 AS SELECT * FROM regular_table
   WHERE col1 IN (SELECT coverage FROM policy_info);
SET CURRENT TEMPORAL BUSINESS_TIME = '2008-01-01';
SELECT * FROM view1;
SELECT * FROM view2;
```

- Set the special register to a past date and issue a query that contains a time period specification. For example:

```
SET CURRENT TEMPORAL BUSINESS_TIME = CURRENT DATE - 1 YEAR;
SELECT * FROM policy_info FOR BUSINESS_TIME AS OF '2008-01-01';
```

### Results

The policy_info table is as follows:

*Table 54. Data in the application-period temporal table (policy_info) after the DELETE statement*

| policy_id | coverage | bus_start | bus_end |
|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 |
| A123 | 14000 | 2008-06-01 | 2008-06-15 |
| A123 | 16000 | 2008-08-15 | 2009-01-01 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 |

- The request for data as of 2008-01-01 queries the policy_info table. The query is implicitly rewritten to:

  ```
  SELECT * FROM policy_info FOR BUSINESS_TIME AS OF '2008-01-01';
  ```

  The query returns:

  ```
  A123, 12000, 2008-01-01, 2008-06-01
  C567, 25000, 2008-01-01, 2009-01-01
  ```

- The query on view1 is implicitly rewritten to:

  ```
  SELECT * FROM view1 FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME;
  ```

  and then to:

  ```
  SELECT policy_id, coverage FROM policy_info
     FOR BUSINESS_TIME AS OF '2008-01-01';
  ```

  The query returns:

  ```
  A123, 12000
  C567, 25000
  ```

  The query on view2 involves a view on a regular table that references an application-period temporal table, causing an implicit relationship between a regular table and the special register. The query is implicitly rewritten to:

  ```
  SELECT * FROM view2 FOR BUSINESS_TIME AS OF CURRENT TEMPORAL BUSINESS_TIME;
  ```

  and then to:

  ```
  SELECT * FROM regular_table WHERE col1 in (SELECT coverage FROM policy_info
     FOR BUSINESS_TIME AS OF '2008-01-01');
  ```

  The SELECT returns rows where col1 values match values in coverage.

- An error is returned because there are multiple time period specifications. The special register was set to a non-null value and the query also specified a time.

## Inserting data into a bitemporal table

Inserting data into a bitemporal table is similar to inserting data into an application-period temporal table.

### About this task

When inserting data into a bitemporal table, include begin and end columns that capture when the row is valid from the perspective of the associated business

applications. This valid period is called the BUSINESS_TIME period. The database manager automatically generates an implicit check constraint that ensures that the begin column of the BUSINESS_TIME period is less than its end column. If a unique constraint or index with BUSINESS_TIME WITHOUT OVERLAPS was created for the table, this ensures that no BUSINESS_TIME periods overlap.

## Procedure

To insert data into a bitemporal table, use the INSERT statement to add data to the table. For example, the following data was inserted on January 31, 2010 (2010-01-31) to the table created in the example in " Creating a bitemporal table".

```
INSERT INTO policy_info(policy_id, coverage, bus_start, bus_end)
   VALUES('A123',12000,'2008-01-01','2008-07-01');

INSERT INTO policy_info(policy_id, coverage, bus_start, bus_end)
   VALUES('A123',16000,'2008-07-01','2009-01-01');

INSERT INTO policy_info(policy_id, coverage, bus_start, bus_end)
   VALUES('B345',18000,'2008-01-01','2009-01-01');

INSERT INTO policy_info(policy_id, coverage, bus_start, bus_end)
   VALUES('C567',20000,'2008-01-01','2009-01-01');
```

## Results

The `policy_info` table now contains the following insurance coverage data. The `sys_start`, `sys_end`, and `ts_id` column entries are generated by the database manager. The begin-column of a period is inclusive, while the end-column is exclusive, meaning that the row with a `bus_end` value of 2008-07-01 does not have a BUSINESS_TIME period overlap with the row that contains a `bus_start` value of 2008-07-01.

*Table 55. Data added to a bitemporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

The `hist_policy_info` history table remains empty because no history rows are generated by an insert.

*Table 56. History table (hist_policy_info) after insert*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| | | | | | | |

# Updating data in a bitemporal table

Updating data in a bitemporal table results in rows that are added to its associated history table and can potentially result in rows that are added to the bitemporal table itself.

## About this task

In addition to the regular UPDATE statement, bitemporal tables also support time range updates where the UPDATE statement includes the FOR PORTION OF BUSINESS_TIME clause. A row is a candidate for updating if its period-begin column, period-end column, or both fall within the range specified in the FOR PORTION OF BUSINESS_TIME clause. Any existing impacted rows are copied to the history table before they are updated.

## Procedure

To update data in a bitemporal table, use the UPDATE statement to change data rows. For example, it was discovered that there are some errors in the insurance coverage levels for two customers and the following data was updated on February 28, 2011 (2011-02-28) in the example table that had data added in the " Inserting data into a bitemporal table" topic.
The following table is the original `policy_info` table data.

*Table 57. Original data in the bitemporal table (policy_info)*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

Updating a bitemporal table by using the FOR PORTION OF BUSINESS_TIME clause causes rows to be changed and can result in rows that are inserted when the existing time period for rows that are updated is not fully contained within the range specified in the UPDATE statement.

- The coverage for policy B345 actually started on March 1, 2008 (2008-03-01):

```
UPDATE policy_info
  SET bus_start='2008-03-01'
  WHERE policy_id = 'B345'
  AND coverage = 18000;
```

  The update to policy B345 uses a regular UPDATE statement. There is only one row in the `policy_info` table for `policy_id` B345, so there are no potential BUSINESS_TIME periods overlaps. As a result the following things occur:

  1. The `bus_start` column value is updated to 2008-03-01. Note that updates to a BUSINESS_TIME period column cannot include the FOR PORTION OF BUSINESS_TIME clause.

2. The database manager updates the sys_start and ts_id values to the date of the update.
3. The original row is moved to the history table. The database manager updates the sys_end value to the date of the update.

The following tables show the update for policy B345.

*Table 58. Bitemporal table (policy_info) after policy B345 update*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

*Table 59. History table (hist_policy_info) after policy B345 update*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |

- The coverage for policy C567 should be 25000 for the year 2008:

```
UPDATE policy_info
   FOR PORTION OF BUSINESS_TIME FROM '2008-01-01' TO '2009-01-01'
   SET coverage = 25000
   WHERE policy_id = 'C567';
```

The update to policy C567 applies to the BUSINESS_TIME period from 2008-01-01 to 2009-01-01. There is only one row in the policy_info table for policy_id C567 that includes that time period. The BUSINESS_TIME period is fully contained within the bus_start and bus_end column values for that row. As a result the following things occur:

1. The coverage value for the row with policy_id C567 is updated to 25000.
2. The bus_start and bus_end column values are unchanged.
3. The database manager updates the sys_start and ts_id values to the date of the update.
4. The original row is moved to the history table. The database manager updates the sys_end value to the date of the update.

The following tables show the update for policy C567.

*Table 60. Bitemporal table (policy_info) after policy C567 update*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |

*Table 60. Bitemporal table (policy_info) after policy C567 update (continued)*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 9999-12-30-00.00.00.000000000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 61. History table (hist_policy_info) after policy C567 update*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |

- The coverage for policy A123 shows an increase from 12000 to 16000 on July 7 (2008-07-01), but an earlier increase to 14000 is missing:

```
UPDATE policy_info
   FOR PORTION OF BUSINESS_TIME FROM '2008-06-01' TO '2008-08-01'
   SET coverage = 14000
   WHERE policy_id = 'A123';
```

  The update to policy A123 applies to the BUSINESS_TIME period from 2008-06-01 to 2008-08-01. There are two rows in the `policy_info` table for `policy_id` A123 that include part of the update time period.

  1. The BUSINESS_TIME period is partially contained in the row that has a `bus_start` value of 2008-01-01 and a `bus_end` value of 2008-07-01. This row overlaps the beginning of the specified period because the earliest time value in the BUSINESS_TIME period is greater than the rows `bus_start` value, but less than its `bus_end` value.
  2. The BUSINESS_TIME period is partially contained in the row that has a `bus_start` value of 2008-07-01 and a `bus_end` value of 2009-01-01. This row overlaps the end of the specified period because the latest time value in the BUSINESS_TIME period is greater than the rows `bus_start` value, but less than its `bus_end` value.

  As a result the following things occur:

  1. When the `bus_end` value overlaps the beginning of the specified period, the row is updated to the new coverage value of 14000. In this updated row, the `bus_start` value is set to 2008-06-01 which is the begin value of the UPDATE specified period, and the `bus_end` value is unchanged. An additional row is inserted with the original values from the row, except that the `bus_end` value is set to 2008-06-01. This new row reflects the BUSINESS_TIME period when coverage was 12000. The `sys_start`, `sys_end`, and `ts_id` column entries are generated by the database manager.
  2. When the `bus_start` value overlaps the end of the specified period, the row is updated to the new coverage value of 14000. In this updated row, the `bus_start` value is unchanged and the `bus_end` value is set to 2008-08-01

which is the end value of the UPDATE specified period. An additional row is inserted with the original values from the row, except that the `bus_start` value is set to 2008-08-01. This new row reflects the BUSINESS_TIME period when coverage was 16000. The `sys_start`, `sys_end`, and `ts_id` column entries are generated by the database manager.

3. The original rows are moved to the history table. The database manager updates the `sys_end` value to the date of the update.

The following tables show the update for policy A123.

*Table 62. Bitemporal table (policy_info) after policy A123 update*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 63. History table (hist_policy_info) after policy A123 update*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |

## Deleting data from a bitemporal table

Deleting data from a bitemporal table results in rows that are deleted from the table, rows that are added to its associated history table and can potentially result in new rows that are inserted into the bitemporal table itself.

## About this task

In addition to the regular DELETE statement, bitemporal tables also support time range deletes where the DELETE statement includes the FOR PORTION OF BUSINESS_TIME clause. A row is a candidate for deletion if its period-begin column, period-end column, or both falls within the range specified in the FOR PORTION OF BUSINESS_TIME clause. Any existing impacted rows are copied to the history table before they are deleted.

## Procedure

To delete data from a bitemporal table, use the DELETE FROM statement. For example, it was discovered that policy A123 did not have coverage from June 15, 2008 to August 15, 2008. The data was deleted on September 1, 2011 (2011-09-01) from the table that was updated in the " Updating data in a bitemporal table" topic.

```
DELETE FROM policy_info
   FOR PORTION OF BUSINESS_TIME FROM '2008-06-15' TO '2008-08-15'
   WHERE policy_id = 'A123';
```

## Results

The original `policy_info` table and `hist_policy_info` table data is as follows:

*Table 64. Data in the bitemporal table (policy_info) before the DELETE statement*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 65. Data in the history table (hist_policy_info) before the DELETE statement*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |

*Table 65. Data in the history table (hist_policy_info) before the DELETE statement (continued)*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |

Deleting data from a bitemporal table by using the FOR PORTION OF BUSINESS_TIME clause causes rows to be deleted and can result in rows that are inserted when the time period for a row covers a portion of the range specified in the DELETE FROM statement. Deleting data related to policy A123 applies to the BUSINESS_TIME period from 2008-06-15 to 2008-08-15. There are three rows in the `policy_info` table for `policy_id` A123 that include all or part of that time period.

As a result, the following things occur:

- There is one row where the BUSINESS_TIME period in the DELETE FROM statement covers the entire time period for a row. The row with a `bus_start` value of 2008-07-01 and a `bus_end` value of 2008-08-01 is deleted.
- When only the `bus_end` value falls into the specified period, the row is deleted. A new row is inserted with the original values from the deleted row, except that the `bus_end` value is set to 2008-06-15. The `sys_start`, `sys_end`, and `ts_id` column entries are generated by the database manager.
- When only the `bus_start` value falls into the specified period, the row is deleted. A new row is inserted with the original values from the deleted row, except that the `bus_start` value is set to 2008-08-15. The `sys_start`, `sys_end`, and `ts_id` column entries are generated by the database manager.
- The original rows are moved to the history table. The database manager updates the `sys_end` value to the date of the delete.

*Table 66. Data in the bitemporal table (policy_info) after the DELETE statement*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| A123 | 14000 | 2008-06-01 | 2008-06-15 | 2011-09-01-12.18.22.959254000000 | 9999-12-30-00.00.00.000000000000 | 2011-09-01-12.18.22.959254000000 |
| A123 | 16000 | 2008-08-15 | 2009-01-01 | 2011-09-01-12.18.22.959254000000 | 9999-12-30-00.00.00.000000000000 | 2011-09-01-12.18.22.959254000000 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.649592000000 |

*Table 67. History table (hist_policy_info) after DELETE statement*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.649592000000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 | 2011-02-28-09.10.12.649592000000 | 2011-09-01-12.18.22.959254000000 | 2011-09-01-12.18.22.959254000000 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 | 2011-02-28-09.10.12.649592000000 | 2011-09-01-12.18.22.959254000000 | 2011-09-01-12.18.22.959254000000 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 | 2011-02-28-09.10.12.649592000000 | 2011-09-01-12.18.22.959254000000 | 2011-09-01-12.18.22.959254000000 |

# Querying bitemporal data

Querying a bitemporal table can return results for a specified time period. Those results can include current values, previous historic values, and future values.

## About this task

When querying a bitemporal table, you can include FOR BUSINESS_TIME, FOR SYSTEM_TIME, or both in the FROM clause. Using these time period specifications, you can query the current, past, and future state of your data. Time periods are specified as follows:

**AS OF** *value1*
: Includes all the rows where the begin value for the period is less than or equal to *value1* and the end value for the period is greater than *value1*. This enables you to query your data as of a certain point in time.

**FROM** *value1* **TO** *value2*
: Includes all the rows where the begin value for the period is equal to or greater than *value1* and the end value for the period is less than *value2*. This means that the begin time is included in the period, but the end time is not.

**BETWEEN** *value1* **AND** *value2*
: Includes all the rows where any time period overlaps any point in time between *value1* and *value2*. A row is returned if the begin value for the period is less than or equal to *value2* and the end value for the period is greater than *value1*.

See the following section for some sample queries.

## Procedure

To query a bitemporal table, use the SELECT statement. For example, each of the following queries requests policy information for `policy_id` A123 from the result tables in the " Deleting data from a bitemporal table" topic. Each query uses a variation of the time period specification.
The `policy_info` table and its associated history table are as follows:

*Table 68. Bitemporal table: policy_info*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-06-01 | 2011-02-28-09.10.12.64959200000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.64959200000 |
| A123 | 14000 | 2008-06-01 | 2008-06-15 | 2011-09-01-12.18.22.959254000000 | 9999-12-30-00.00.00.000000000000 | 2011-09-01-12.18.22.959254000000 |
| A123 | 16000 | 2008-08-15 | 2009-01-01 | 2011-09-01-12.18.22.959254000000 | 9999-12-30-00.00.00.000000000000 | 2011-09-01-12.18.22.959254000000 |
| B345 | 18000 | 2008-03-01 | 2009-01-01 | 2011-02-28-09.10.12.64959200000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.64959200000 |
| C567 | 25000 | 2008-01-01 | 2009-01-01 | 2011-02-28-09.10.12.64959200000 | 9999-12-30-00.00.00.000000000000 | 2011-02-28-09.10.12.64959200000 |

*Table 69. History table: hist_policy_info*

| policy_id | coverage | bus_start | bus_end | sys_start | sys_end | ts_id |
|---|---|---|---|---|---|---|
| A123 | 12000 | 2008-01-01 | 2008-07-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.64959200000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 16000 | 2008-07-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.64959200000 | 2010-01-31-22.31.33.495925000000 |
| B345 | 18000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.64959200000 | 2010-01-31-22.31.33.495925000000 |
| C567 | 20000 | 2008-01-01 | 2009-01-01 | 2010-01-31-22.31.33.495925000000 | 2011-02-28-09.10.12.64959200000 | 2010-01-31-22.31.33.495925000000 |
| A123 | 14000 | 2008-06-01 | 2008-07-01 | 2011-02-28-09.10.12.64959200000 | 2011-09-01-12.18.22.959254000000 | 2011-09-01-12.18.22.959254000000 |
| A123 | 14000 | 2008-07-01 | 2008-08-01 | 2011-02-28-09.10.12.64959200000 | 2011-09-01-12.18.22.959254000000 | 2011-09-01-12.18.22.959254000000 |
| A123 | 16000 | 2008-08-01 | 2009-01-01 | 2011-02-28-09.10.12.64959200000 | 2011-09-01-12.18.22.959254000000 | 2011-09-01-12.18.22.959254000000 |

- Query with no time period specification. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   where policy_id = 'A123'
```

This query returns three rows. The SELECT statement queries only the `policy_info` table. The history table is not queried because FOR SYSTEM_TIME was not specified.

```
A123, 12000, 2008-01-01, 2008-06-01
A123, 14000, 2008-06-01, 2008-06-15
A123, 16000, 2008-08-15, 2009-01-01
```

- Query with FOR SYSTEM_TIME FROM...TO specified. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   FOR SYSTEM_TIME FROM
      '0001-01-01-00.00.00.000000' TO '9999-12-30-00.00.00.000000000000'
   where policy_id = 'A123'
```

This query returns eight rows. The SELECT statement queries both the `policy_info` and the `hist_policy_info` tables.

```
A123, 12000, 2008-01-01, 2008-06-01
A123, 14000, 2008-06-01, 2008-06-15
A123, 16000, 2008-08-15, 2009-01-01
A123, 12000, 2008-01-01, 2008-07-01
A123, 16000, 2008-07-01, 2009-01-01
A123, 14000, 2008-06-01, 2008-07-01
A123, 14000, 2008-07-01, 2008-08-01
A123, 16000, 2008-08-01, 2009-01-01
```

- Query with FOR BUSINESS_TIME AS OF specified. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   FOR BUSINESS_TIME AS OF '2008-07-15'
   where policy_id = 'A123'
```

This query does not return any rows. The SELECT statement queries only the `policy_info` table and there are no rows for A123 where the begin value for the period is less than or equal to *2008-07-15* and the end value for the period is greater than *2008-07-15*. Policy A123 had no coverage on 2008-07-15. The history table is not queried because FOR SYSTEM_TIME was not specified.

- Query with FOR BUSINESS_TIME AS OF and FOR SYSTEM_TIME FROM...TO specified. For example:

```
SELECT policy_id, coverage, bus_start, bus_end
   FROM policy_info
   FOR BUSINESS_TIME AS OF '2008-07-15'
   FOR SYSTEM_TIME FROM
      '0001-01-01-00.00.00.000000' TO '9999-12-30-00.00.00.000000000000'
   where policy_id = 'A123'
```

This query returns two rows. The SELECT queries both the `policy_info` and the `hist_policy_info` tables. The returned rows are found in the history table.

```
A123, 16000, 2008-07-01, 2009-01-01
A123, 14000, 2008-07-01, 2008-08-01
```

# Part 5. Data concurrency

Because many users access and change data in a relational database, the database manager must allow users to make these changes while ensuring that data integrity is preserved.

*Concurrency* refers to the sharing of resources by multiple interactive users or application programs at the same time. The database manager controls this access to prevent undesirable effects, such as:

- **Lost updates**. Two applications, A and B, might both read the same row and calculate new values for one of the columns based on the data that these applications read. If A updates the row and then B also updates the row, A's update lost.
- **Access to uncommitted data**. Application A might update a value, and B might read that value before it is committed. Then, if A backs out of that update, the calculations performed by B might be based on invalid data.
- **Non-repeatable reads**. Application A might read a row before processing other requests. In the meantime, B modifies or deletes the row and commits the change. Later, if A attempts to read the original row again, it sees the modified row or discovers that the original row has been deleted.
- **Phantom reads**. Application A might execute a query that reads a set of rows based on some search criterion. Application B inserts new data or updates existing data that would satisfy application A's query. Application A executes its query again, within the same unit of work, and some additional (“phantom”) values are returned.

Concurrency is not an issue for global temporary tables, because they are available only to the application that declares or creates them.

# Chapter 40. Isolation levels

The *isolation level* that is associated with an application process determines the degree to which the data that is being accessed by that process is locked or isolated from other concurrently executing processes. The isolation level is in effect for the duration of a unit of work.

The isolation level of an application process therefore specifies:

- The degree to which rows that are read or updated by the application are available to other concurrently executing application processes
- The degree to which the update activity of other concurrently executing application processes can affect the application

The isolation level for static SQL statements is specified as an attribute of a package and applies to the application processes that use that package. The isolation level is specified during the program preparation process by setting the ISOLATION bind or precompile option. For dynamic SQL statements, the default isolation level is the isolation level that was specified for the package preparing the statement. Use the SET CURRENT ISOLATION statement to specify a different isolation level for dynamic SQL statements that are issued within a session. For more information, see "CURRENT ISOLATION special register". For both static SQL statements and dynamic SQL statements, the *isolation-clause* in a *select-statement* overrides both the special register (if set) and the bind option value. For more information, see "Select-statement".

Isolation levels are enforced by locks, and the type of lock that is used limits or prevents access to the data by concurrent application processes. Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.

The database manager supports three general categories of locks:

**Share (S)**
Under an S lock, concurrent application processes are limited to read-only operations on the data.

**Update (U)**
Under a U lock, concurrent application processes are limited to read-only operations on the data, if these processes have not declared that they might update a row. The database manager assumes that the process currently looking at a row might update it.

**Exclusive (X)**
Under an X lock, concurrent application processes are prevented from accessing the data in any way. This does not apply to application processes with an isolation level of uncommitted read (UR), which can read but not modify the data.

Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed by an application process during a unit of work is not changed by any other application process until the unit of work is complete.

The database manager supports four isolation levels.

- "Repeatable read (RR)"
- "Read stability (RS)"
- "Cursor stability (CS)" on page 407
- "Uncommitted read (UR)" on page 408

**Note:** Some host database servers support the *no commit (NC)* isolation level. On other database servers, this isolation level behaves like the uncommitted read isolation level.

A detailed description of each isolation level follows, in decreasing order of performance impact, but in increasing order of the care that is required when accessing or updating data.

## Repeatable read (RR)

The *repeatable read* isolation level locks all the rows that an application references during a unit of work (UOW). If an application issues a SELECT statement twice within the same unit of work, the same result is returned each time. Under RR, lost updates, access to uncommitted data, non-repeatable reads, and phantom reads are not possible.

Under RR, an application can retrieve and operate on the rows as many times as necessary until the UOW completes. However, no other application can update, delete, or insert a row that would affect the result set until the UOW completes. Applications running under the RR isolation level cannot see the uncommitted changes of other applications. This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

Every referenced row is locked, not just the rows that are retrieved. For example, if you scan 10 000 rows and apply predicates to them, locks are held on all 10 000 rows, even if, say, only 10 rows qualify. Another application cannot insert or update a row that would be added to the list of rows referenced by a query if that query were to be executed again. This prevents phantom reads.

Because RR can acquire a considerable number of locks, this number might exceed limits specified by the **locklist** and **maxlocks** database configuration parameters. To avoid lock escalation, the optimizer might elect to acquire a single table-level lock for an index scan, if it appears that lock escalation is likely. If you do not want table-level locking, use the read stability isolation level.

While evaluating referential constraints, the DB2 server might occasionally upgrade the isolation level used on scans of the foreign table to RR, regardless of the isolation level that was previously set by the user. This results in additional locks being held until commit time, which increases the likelihood of a deadlock or a lock timeout. To avoid these problems, create an index that contains only the foreign key columns, which the referential integrity scan can use instead.

## Read stability (RS)

The *read stability* isolation level locks only those rows that an application retrieves during a unit of work. RS ensures that any qualifying row read during a UOW cannot be changed by other application processes until the UOW completes, and that any change to a row made by another application process cannot be read until the change is committed by that process. Under RS, access to uncommitted data

and non-repeatable reads are not possible. However, phantom reads are possible. Phantom reads might also be introduced by concurrent updates to rows where the old value did not satisfy the search condition of the original application but the new updated value does.

For example, a phantom row can occur in the following situation:

1. Application process P1 reads the set of rows n that satisfy some search condition.
2. Application process P2 then inserts one or more rows that satisfy the search condition and commits those new inserts.
3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.

In a DB2 pureScale environment, an application running at this isolation level might reject a previously committed row value if the row is updated concurrently on a different member. To override this behavior, specify the `WAIT_FOR_OUTCOME` option.

This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

The RS isolation level provides both a high degree of concurrency and a stable view of the data. To that end, the optimizer ensures that table-level locks are not obtained until lock escalation occurs.

The RS isolation level is suitable for an application that:

- Operates in a concurrent environment
- Requires qualifying rows to remain stable for the duration of a unit of work
- Does not issue the same query more than once during a unit of work, or does not require the same result set when a query is issued more than once during a unit of work

## Cursor stability (CS)

The *cursor stability* isolation level locks any row being accessed during a transaction while the cursor is positioned on that row. This lock remains in effect until the next row is fetched or the transaction terminates. However, if any data in the row was changed, the lock is held until the change is committed.

Under this isolation level, no other application can update or delete a row while an updatable cursor is positioned on that row. Under CS, access to the uncommitted data of other applications is not possible. However, non-repeatable reads and phantom reads are possible.

CS is the default isolation level. It is suitable when you want maximum concurrency and need to see only committed data.

In a DB2 pureScale environment, an application running at this isolation level may return or reject a previously committed row value if the row is concurrently updated on a different member. The `WAIT FOR OUTCOME` option of the concurrent access resolution setting can be used to override this behavior.

**Note:** Under the *currently committed* semantics introduced in Version 9.7, only committed data is returned, as was the case previously, but now readers do not

wait for updaters to release row locks. Instead, readers return data that is based on the currently committed version; that is, data prior to the start of the write operation.

## Uncommitted read (UR)

The *uncommitted read* isolation level allows an application to access the uncommitted changes of other transactions. Moreover, UR does not prevent another application from accessing a row that is being read, unless that application is attempting to alter or drop the table.

Under UR, access to uncommitted data, non-repeatable reads, and phantom reads are possible. This isolation level is suitable if you run queries against read-only tables, or if you issue SELECT statements only, and seeing data that has not been committed by other applications is not a problem.

UR works differently for read-only and updatable cursors.

- Read-only cursors can access most of the uncommitted changes of other transactions.
- Tables, views, and indexes that are being created or dropped by other transactions are not available while the transaction is processing. Any other changes by other transactions can be read before they are committed or rolled back. Updatable cursors operating under UR behave as though the isolation level were CS.

If an uncommitted read application uses ambiguous cursors, it might use the CS isolation level when it runs. The ambiguous cursors can be escalated to CS if the value of the BLOCKING option on the **PREP** or **BIND** command is UNAMBIG (the default). To prevent this escalation:

- Modify the cursors in the application program to be unambiguous. Change the SELECT statements to include the FOR READ ONLY clause.
- Let the cursors in the application program remain ambiguous, but precompile the program or bind it with the BLOCKING ALL and STATICREADONLY YES options to enable the ambiguous cursors to be treated as read-only when the program runs.

## Comparison of isolation levels

Table 70 summarizes the supported isolation levels.

*Table 70. Comparison of isolation levels*

| | UR | CS | RS | RR |
|---|---|---|---|---|
| Can an application see uncommitted changes made by other application processes? | Yes | No | No | No |
| Can an application update uncommitted changes made by other application processes? | No | No | No | No |
| Can the re-execution of a statement be affected by other application processes? [1] | Yes | Yes | Yes | No [2] |
| Can updated rows be updated by other application processes? [3] | No | No | No | No |
| Can updated rows be read by other application processes that are running at an isolation level other than UR? | No | No | No | No |

*Table 70. Comparison of isolation levels (continued)*

| | UR | CS | RS | RR |
|---|---|---|---|---|
| Can updated rows be read by other application processes that are running at the UR isolation level? | Yes | Yes | Yes | Yes |
| Can accessed rows be updated by other application processes? [4] | Yes | Yes | No | No |
| Can accessed rows be read by other application processes? | Yes | Yes | Yes | Yes |
| Can the current row be updated or deleted by other application processes? [5] | Yes/No [6] | Yes/No [6] | No | No |

**Note:**

1. An example of the *phantom read phenomenon* is as follows: Unit of work UW1 reads the set of *n* rows that satisfies some search condition. Unit of work UW2 inserts one or more rows that satisfy the same search condition and then commits. If UW1 subsequently repeats its read with the same search condition, it sees a different result set: the rows that were read originally plus the rows that were inserted by UW2.
2. If your label-based access control (LBAC) credentials change between reads, results for the second read might be different because you have access to different rows.
3. The isolation level offers no protection to the application if the application is both reading from and writing to a table. For example, an application opens a cursor on a table and then performs an insert, update, or delete operation on the same table. The application might see inconsistent data when more rows are fetched from the open cursor.
4. An example of the *non-repeatable read phenomenon* is as follows: Unit of work UW1 reads a row. Unit of work UW2 modifies that row and commits. If UW1 subsequently reads that row again, it might see a different value.
5. An example of the *dirty read phenomenon* is as follows: Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 commits. If UW1 subsequently rolls the changes back, UW2 has read nonexisting data.
6. Under UR or CS, if the cursor is not updatable, the current row can be updated or deleted by other application processes in some cases. For example, buffering might cause the current row at the client to be different from the current row at the server. Moreover, when using currently committed semantics under CS, a row that is being read might have uncommitted updates pending. In this case, the currently committed version of the row is always returned to the application.

## Summary of isolation levels

Table 71 lists the concurrency issues that are associated with different isolation levels.

*Table 71. Summary of isolation levels*

| Isolation level | Access to uncommitted data | Non-repeatable reads | Phantom reads |
|---|---|---|---|
| Repeatable read (RR) | Not possible | Not possible | Not possible |
| Read stability (RS) | Not possible | Not possible | Possible |
| Cursor stability (CS) | Not possible | Possible | Possible |
| Uncommitted read (UR) | Possible | Possible | Possible |

The isolation level affects not only the degree of isolation among applications but also the performance characteristics of an individual application, because the

processing and memory resources that are required to obtain and free locks vary with the isolation level. The potential for deadlocks also varies with the isolation level. Table 72 provides a simple heuristic to help you choose an initial isolation level for your application.

*Table 72. Guidelines for choosing an isolation level*

| Application type | High data stability required | High data stability not required |
|---|---|---|
| Read-write transactions | RS | CS |
| Read-only transactions | RR or RS | UR |

## Specifying the isolation level

Because the isolation level determines how data is isolated from other processes while the data is being accessed, you should select an isolation level that balances the requirements of concurrency and data integrity.

### About this task

The isolation level that you specify is in effect for the duration of the unit of work (UOW). The following heuristics are used to determine which isolation level will be used when compiling an SQL or XQuery statement:

- For static SQL:
  - If an *isolation-clause* is specified in the statement, the value of that clause is used.
  - If an *isolation-clause* is not specified in the statement, the isolation level that was specified for the package when the package was bound to the database is used.
- For dynamic SQL:
  - If an *isolation-clause* is specified in the statement, the value of that clause is used.
  - If an *isolation-clause* is not specified in the statement, and a SET CURRENT ISOLATION statement has been issued within the current session, the value of the CURRENT ISOLATION special register is used.
  - If an *isolation-clause* is not specified in the statement, and a SET CURRENT ISOLATION statement has not been issued within the current session, the isolation level that was specified for the package when the package was bound to the database is used.
- For static or dynamic XQuery statements, the isolation level of the environment determines the isolation level that is used when the XQuery expression is evaluated.

**Note:** Many commercially-written applications provide a method for choosing the isolation level. Refer to the application documentation for information.

The isolation level can be specified in several different ways.

### Procedure

- **At the statement level:**

  **Note:** Isolation levels for XQuery statements cannot be specified at the statement level.

Use the WITH clause. The WITH clause cannot be used on subqueries. The WITH UR option applies to read-only operations only. In other cases, the statement is automatically changed from UR to CS.

This isolation level overrides the isolation level that is specified for the package in which the statement appears. You can specify an isolation level for the following SQL statements:

- DECLARE CURSOR
- Searched DELETE
- INSERT
- SELECT
- SELECT INTO
- Searched UPDATE

- **For dynamic SQL within the current session:**

  Use the SET CURRENT ISOLATION statement to set the isolation level for dynamic SQL issued within a session. Issuing this statement sets the CURRENT ISOLATION special register to a value that specifies the isolation level for any dynamic SQL statements that are issued within the current session. Once set, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement that is compiled within the session, regardless of which package issued the statement. This isolation level is in effect until the session ends or until the SET CURRENT ISOLATION...RESET statement is issued.

- **At precompile or bind time:**

  For an application written in a supported compiled language, use the ISOLATION option of the **PREP** or **BIND** commands. You can also use the sqlaprep or sqlabndx API to specify the isolation level.

  - If you create a bind file at precompile time, the isolation level is stored in the bind file. If you do not specify an isolation level at bind time, the default is the isolation level that was used during precompilation.
  - If you do not specify an isolation level, the default level of cursor stability (CS) is used.

  To determine the isolation level of a package, execute the following query:

```
select isolation from syscat.packages
  where pkgname = 'pkgname'
    and pkgschema = 'pkgschema'
```

  where *pkgname* is the unqualified name of the package and *pkgschema* is the schema name of the package. Both of these names must be specified in uppercase characters.

- **When working with JDBC or SQLJ at run time:**

  **Note:** JDBC and SQLJ are implemented with CLI on DB2 servers, which means that the `db2cli.ini` settings might affect what is written and run using JDBC and SQLJ.

  To create a package (and specify its isolation level) in SQLJ, use the SQLJ profile customizer (**db2sqljcustomize** command).

- **From CLI or ODBC at run time:**

  Use the **CHANGE ISOLATION LEVEL** command. With DB2 Call-level Interface (CLI), you can change the isolation level as part of the CLI configuration. At run time, use the SQLSetConnectAttr function with the SQL_ATTR_TXN_ISOLATION attribute to set the transaction isolation level for the current connection

referenced by the *ConnectionHandle* argument. You can also use the TXNISOLATION keyword in the `db2cli.ini` file.

- **On database servers that support REXX:**

  When a database is created, multiple bind files that support the different isolation levels for SQL in REXX are bound to the database. Other command line processor (CLP) packages are also bound to the database when a database is created.

  REXX and the CLP connect to a database using the default CS isolation level. Changing this isolation level does not change the connection state.

  To determine the isolation level that is being used by a REXX application, check the value of the SQLISL predefined REXX variable. The value is updated each time that the **CHANGE ISOLATION LEVEL** command executes.

### Results

## Currently committed semantics

Under *currently committed* semantics, only committed data is returned to readers. However, readers do not wait for writers to release row locks. Instead, readers return data that is based on the currently committed version of data: that is, the version of the data before the start of the write operation.

Lock timeouts and deadlocks can occur under the cursor stability (CS) isolation level with row-level locking, especially with applications that are not designed to prevent such problems. Some high-throughput database applications cannot tolerate waiting on locks that are issued during transaction processing. Also, some applications cannot tolerate processing uncommitted data but still require non-blocking behavior for read transactions.

Currently committed semantics are turned on by default for new databases. You do not have to make application changes to take advantage of the new behavior. To override the default behavior, Set the **cur_commit** database configuration parameter to `DISABLED`. Overriding the behavior might be useful, for example, if applications require the blocking of writers to synchronize internal logic. During database upgrade from V9.5 or earlier, the **cur_commit** configuration parameter is set to `DISABLED` to maintain the same behavior as in previous releases. If you want to use currently committed on cursor stability scans, you need to set the **cur_commit** configuration parameter to `ON` after the upgrade.

Currently committed semantics apply only to read-only scans that do not involve catalog tables and internal scans that are used to evaluate or enforce constraints. Because currently committed semantics are decided at the scan level, the access plan of a writer might include currently committed scans. For example, the scan for a read-only subquery can involve currently committed semantics.

Because currently committed semantics obey isolation level semantics, applications running under currently committed semantics continue to respect isolation levels.

Currently committed semantics require increased log space for writers. Additional space is required for logging the first update of a data row during a transaction. This data is required for retrieving the currently committed image of the row. Depending on the workload, this can have an insignificant or measurable impact on the total log space used. The requirement for additional log space does not apply when **cur_commit** database configuration parameter is set to `DISABLED`.

### Restrictions

The following restrictions apply to currently committed semantics:

- The target table object in a section that is to be used for data update or deletion operations does not use currently committed semantics. Rows that are to be modified must be lock protected to ensure that they do not change after they have satisfied any query predicates that are part of the update operation.
- A transaction that makes an uncommitted modification to a row forces the currently committed reader to access appropriate log records to determine the currently committed version of the row. Although log records that are no longer in the log buffer can be physically read, currently committed semantics do not support the retrieval of log files from the log archive. This affects only databases that you configure to use infinite logging.
- The following scans do not use currently committed semantics:
  - Catalog table scans
  - Scans that are used to enforce referential integrity constraints
  - Scans that reference LONG VARCHAR or LONG VARGRAPHIC columns
  - Range-clustered table (RCT) scans
  - Scans that use spatial or extended indexes

### Example

Consider the following scenario, in which deadlocks are avoided by using currently committed semantics. In this scenario, two applications update two separate tables, as shown in step 1, but do not yet commit. Each application then attempts to use a read-only cursor to read from the table that the other application updated, as shown in step 2. These applications are running under the CS isolation level.

| Step | Application A | Application B |
|---|---|---|
| 1 | update T1 set col1 = ? where col2 = ? | update T2 set col1 = ? where col2 = ? |
| 2 | select col1, col3, col4 from T2 where col2 >= ? | select col1, col5, from T1 where col5 = ? and col2 = ? |
| 3 | commit | commit |

Without currently committed semantics, these applications running under the cursor stability isolation level might create a deadlock, causing one of the applications to fail. This happens when each application must read data that is being updated by the other application.

Under currently committed semantics, if one of the applications that is running a query in step 2 requires the data that is being updated by the other application, the first application does not wait for the lock to be released. As a result, a deadlock is impossible. The first application locates and uses the previously committed version of the data instead.

## Option to disregard uncommitted insertions

The **DB2_SKIPINSERTED** registry variable controls whether or not uncommitted data insertions can be ignored for statements that use the cursor stability (CS) or the read stability (RS) isolation level.

Uncommitted insertions are handled in one of two ways, depending on the value of the **DB2_SKIPINSERTED** registry variable.

- When the value is ON, the DB2 server ignores uncommitted insertions, which in many cases can improve concurrency and is the preferred behavior for most applications. Uncommitted insertions are treated as though they had not yet occurred.
- When the value is OFF (the default), the DB2 server waits until the insert operation completes (commits or rolls back) and then processes the data accordingly. This is appropriate in certain cases. For example:
  - Suppose that two applications use a table to pass data between themselves, with the first application inserting data into the table and the second one reading it. The data must be processed by the second application in the order presented, such that if the next row to be read is being inserted by the first application, the second application must wait until the insert operation commits.
  - An application avoids UPDATE statements by deleting data and then inserting a new image of the data.

## Evaluate uncommitted data through lock deferral

To improve concurrency, the database manager in some situations permits the deferral of row locks for CS or RS isolation scans until a row is known to satisfy the predicates of a query.

By default, when row-level locking is performed during a table or index scan, the database manager locks each scanned row whose commitment status is unknown before determining whether the row satisfies the predicates of the query.

To improve the concurrency of such scans, enable the **DB2_EVALUNCOMMITTED** registry variable so that predicate evaluation can occur on uncommitted data. A row that contains an uncommitted update might not satisfy the query, but if predicate evaluation is deferred until after the transaction completes, the row might indeed satisfy the query.

Uncommitted deleted rows are skipped during table scans, and the database manager skips deleted keys during index scans if the **DB2_SKIPDELETED** registry variable is enabled.

The **DB2_EVALUNCOMMITTED** registry variable setting applies at compile time for dynamic SQL or XQuery statements, and at bind time for static SQL or XQuery statements. This means that even if the registry variable is enabled at run time, the lock avoidance strategy is not deployed unless **DB2_EVALUNCOMMITTED** was enabled at bind time. If the registry variable is enabled at bind time but not enabled at run time, the lock avoidance strategy is still in effect. For static SQL or XQuery statements, if a package is rebound, the registry variable setting that is in effect at bind time is the setting that applies. An implicit rebind of static SQL or XQuery statements will use the current setting of the **DB2_EVALUNCOMMITTED** registry variable.

### Applicability of evaluate uncommitted for different access plans

*Table 73. RID Index Only Access*

| Predicates | Evaluate Uncommitted |
|---|---|
| None | No |

*Table 73. RID Index Only Access (continued)*

| Predicates | Evaluate Uncommitted |
|---|---|
| SARGable | Yes |

*Table 74. Data Only Access (relational or deferred RID list)*

| Predicates | Evaluate Uncommitted |
|---|---|
| None | No |
| SARGable | Yes |

*Table 75. RID Index + Data Access*

| Predicates | | Evaluate Uncommitted | |
|---|---|---|---|
| **Index** | **Data** | **Index access** | **Data access** |
| None | None | No | No |
| None | SARGable | No | No |
| SARGable | None | Yes | No |
| SARGable | SARGable | Yes | No |

*Table 76. Block Index + Data Access*

| Predicates | | Evaluate Uncommitted | |
|---|---|---|---|
| **Index** | **Data** | **Index access** | **Data access** |
| None | None | No | No |
| None | SARGable | No | Yes |
| SARGable | None | Yes | No |
| SARGable | SARGable | Yes | Yes |

## Example

The following example provides a comparison between the default locking behavior and the evaluate uncommitted behavior. The table is the ORG table from the SAMPLE database.

```
DEPTNUMB DEPTNAME       MANAGER DIVISION   LOCATION
-------- -------------- ------- ---------- -------------
      10 Head Office        160 Corporate  New York
      15 New England         50 Eastern    Boston
      20 Mid Atlantic        10 Eastern    Washington
      38 South Atlantic      30 Eastern    Atlanta
      42 Great Lakes        100 Midwest    Chicago
      51 Plains             140 Midwest    Dallas
      66 Pacific            270 Western    San Francisco
      84 Mountain           290 Western    Denver
```

The following transactions occur under the default cursor stability (CS) isolation level.

*Table 77. Transactions against the ORG table under the CS isolation level*

| SESSION 1 | SESSION 2 |
|---|---|
| `connect to sample` | `connect to sample` |

*Table 77. Transactions against the ORG table under the CS isolation level (continued)*

| SESSION 1 | SESSION 2 |
|---|---|
| `+c update org set deptnumb=5 where manager=160` | |
| | `select * from org where deptnumb >= 10` |

The uncommitted UPDATE statement in Session 1 holds an exclusive lock on the first row in the table, preventing the query in Session 2 from returning a result set, even though the row being updated in Session 1 does not currently satisfy the query in Session 2. The CS isolation level specifies that any row that is accessed by a query must be locked while the cursor is positioned on that row. Session 2 cannot obtain a lock on the first row until Session 1 releases its lock.

Waiting for a lock in Session 2 can be avoided by using the evaluate uncommitted feature, which first evaluates the predicate and then locks the row. As such, the query in Session 2 would not attempt to lock the first row in the table, thereby increasing application concurrency. Note that this also means that predicate evaluation in Session 2 would occur with respect to the uncommitted value of `deptnumb=5` in Session 1. The query in Session 2 would omit the first row in its result set, despite the fact that a rollback of the update in Session 1 would satisfy the query in Session 2.

If the order of operations were reversed, concurrency could still be improved with the evaluate uncommitted feature. Under default locking behavior, Session 2 would first acquire a row lock prohibiting the searched UPDATE in Session 1 from executing, even though the Session 1 UPDATE statement would not change the row that is locked by the Session 2 query. If the searched UPDATE in Session 1 first attempted to examine rows and then locked them only if they qualified, the Session 1 query would be non-blocking.

## Restrictions

- The **DB2_EVALUNCOMMITTED** registry variable must be enabled.
- The isolation level must be CS or RS.
- Row-level locking is in effect.
- SARGable evaluation predicates exist.
- Evaluate uncommitted is not applicable to scans on the system catalog tables.
- For multidimensional clustering (MDC) or insert time clustering (ITC) tables, block-level locking can be deferred for an index scan; however, block-level locking cannot be deferred for table scans.
- Lock deferral will not occur on a table that is executing an inplace table reorganization.
- For Iscan-Fetch plans, row-level locking is not deferred to the data access; rather, the row is locked during index access before moving to the row in the table.
- Deleted rows are unconditionally skipped during table scans, but deleted index keys are skipped only if the **DB2_SKIPDELETED** registry variable is enabled.

# Chapter 41. Locks and concurrency control

To provide concurrency control and prevent uncontrolled data access, the database manager places locks on buffer pools, tables, data partitions, table blocks, or table rows.

A *lock* associates a database manager resource with an application, called the *lock owner*, to control how other applications access the same resource.

The database manager uses row-level locking or table-level locking, as appropriate, based on:

- The isolation level specified at precompile time or when an application is bound to the database. The isolation level can be one of the following:
  - Uncommitted read (UR)
  - Cursor stability (CS)
  - Read stability (RS)
  - Repeatable read (RR)

  The different isolation levels are used to control access to uncommitted data, prevent lost updates, allow non-repeatable reads of data, and prevent phantom reads. To minimize performance impact, use the minimum isolation level that satisfies your application needs.
- The access plan selected by the optimizer. Table scans, index scans, and other methods of data access each require different types of access to the data.
- The LOCKSIZE attribute for the table. The LOCKSIZE clause on the ALTER TABLE statement indicates the granularity of the locks that are used when the table is accessed. The choices are: ROW for row locks, TABLE for table locks, or BLOCKINSERT for block locks on multidimensional clustering (MDC) tables only. When the BLOCKINSERT clause is used on an MDC table, row-level locking is performed, except during an insert operation, when block-level locking is done instead. Use the ALTER TABLE...LOCKSIZE BLOCKINSERT statement for MDC tables when transactions will be performing large inserts into disjointed cells. Use the ALTER TABLE...LOCKSIZE TABLE statement for read-only tables. This reduces the number of locks that are required for database activity. For partitioned tables, table locks are first acquired and then data partition locks are acquired, as dictated by the data that will be accessed.
- The amount of memory devoted to locking, which is controlled by the **locklist** database configuration parameter. If the lock list fills up, performance can degrade because of lock escalations and reduced concurrency among shared objects in the database. If lock escalations occur frequently, increase the value of **locklist**, **maxlocks**, or both. To reduce the number of locks that are held at one time, ensure that transactions commit frequently.

A buffer pool lock (exclusive) is set whenever a buffer pool is created, altered, or dropped. You might encounter this type of lock when collecting system monitoring data. The name of the lock is the identifier (ID) for the buffer pool itself.

In general, row-level locking is used unless one of the following is true:

- The isolation level is uncommitted read
- The isolation level is repeatable read and the access plan requires a scan with no index range predicates

- The table LOCKSIZE attribute is TABLE
- The lock list fills up, causing lock escalation
- An explicit table lock has been acquired through the LOCK TABLE statement, which prevents concurrent application processes from changing or using a table

In the case of an MDC table, block-level locking is used instead of row-level locking when:

- The table LOCKSIZE attribute is BLOCKINSERT
- The isolation level is repeatable read and the access plan involves predicates
- A searched update or delete operation involves predicates on dimension columns only

The duration of row locking varies with the isolation level being used:

- UR scans: No row locks are held unless row data is changing.
- CS scans: Row locks are generally held only while the cursor is positioned on the row. Note that in some cases, locks might not be held at all during a CS scan.
- RS scans: Qualifying row locks are held only for the duration of the transaction.
- RR scans: All row locks are held for the duration of the transaction.

## Lock granularity

If one application holds a lock on a database object, another application might not be able to access that object. For this reason, row-level locks, which minimize the amount of data that is locked and therefore inaccessible, are better for maximum concurrency than block-level, data partition-level, or table-level locks.

However, locks require storage and processing time, so a single table lock minimizes lock overhead.

The LOCKSIZE clause of the ALTER TABLE statement specifies the granularity of locks at the row, data partition, block, or table level. Row locks are used by default. Use of this option in the table definition does not prevent normal lock escalation from occurring.

The ALTER TABLE statement specifies locks globally, affecting all applications and users that access that table. Individual applications might use the LOCK TABLE statement to specify table locks at an application level instead.

A permanent table lock defined by the ALTER TABLE statement might be preferable to a single-transaction table lock using the LOCK TABLE statement if:

- The table is read-only, and will always need only S locks. Other users can also obtain S locks on the table.
- The table is usually accessed by read-only applications, but is sometimes accessed by a single user for brief maintenance, and that user requires an X lock. While the maintenance program is running, read-only applications are locked out, but in other circumstances, read-only applications can access the table concurrently with a minimum of locking overhead.

For a multidimensional clustering (MDC) table, you can specify BLOCKINSERT with the LOCKSIZE clause in order to use block-level locking during insert operations only. When BLOCKINSERT is specified, row-level locking is performed for all other operations, but only minimally for insert operations. That is,

block-level locking is used during the insertion of rows, but row-level locking is used to lock the next key if repeatable read (RR) scans are encountered in the record ID (RID) indexes as they are being updated. BLOCKINSERT locking might be beneficial when:

- There are multiple transactions doing mass insertions into separate cells
- Concurrent insertions into the same cell by multiple transactions is not occurring, or it is occurring with enough data inserted per cell by each of the transactions that the user is not concerned that each transaction will insert into separate blocks

## Lock attributes

Database manager locks have several basic attributes.

These attributes include the following:

**Mode** The type of access allowed for the lock owner, as well as the type of access allowed for concurrent users of the locked object. It is sometimes referred to as the *state* of the lock.

**Object**
The resource being locked. The only type of object that you can lock explicitly is a table. The database manager also sets locks on other types of resources, such as rows and table spaces. Block locks can also be set for multidimensional clustering (MDC) or insert time clustering (ITC) tables, and data partition locks can be set for partitioned tables. The object being locked determines the *granularity* of the lock.

**Lock count**
The length of time during which a lock is held. The isolation level under which a query runs affects the lock count.

Table 78 lists the lock modes and describes their effects, in order of increasing control over resources.

*Table 78. Lock Mode Summary*

| Lock Mode | Applicable Object Type | Description |
|---|---|---|
| **IN (Intent None)** | Table spaces, blocks, tables, data partitions | The lock owner can read any data in the object, including uncommitted data, but cannot update any of it. Other concurrent applications can read or update the table. |
| **IS (Intent Share)** | Table spaces, blocks, tables, data partitions | The lock owner can read data in the locked table, but cannot update this data. Other applications can read or update the table. |
| **IX (Intent Exclusive)** | Table spaces, blocks, tables, data partitions | The lock owner and concurrent applications can read and update data. Other concurrent applications can both read and update the table. |
| **NS (Scan Share)** | Rows | The lock owner and all concurrent applications can read, but not update, the locked row. This lock is acquired on rows of a table, instead of an S lock, where the isolation level of the application is either RS or CS. |
| **NW (Next Key Weak Exclusive)** | Rows | When a row is inserted into an index, an NW lock is acquired on the next row. This occurs only if the next row is currently locked by an RR scan. The lock owner can read but not update the locked row. This lock mode is similar to an X lock, except that it is also compatible with NS locks. |

*Table 78. Lock Mode Summary (continued)*

| Lock Mode | Applicable Object Type | Description |
|---|---|---|
| **S (Share)** | Rows, blocks, tables, data partitions | The lock owner and all concurrent applications can read, but not update, the locked data. |
| **SIX (Share with Intent Exclusive)** | Tables, blocks, data partitions | The lock owner can read and update data. Other concurrent applications can read the table. |
| **U (Update)** | Rows, blocks, tables, data partitions | The lock owner can update data. Other units of work can read the data in the locked object, but cannot update it. |
| **X (Exclusive)** | Rows, blocks, tables, buffer pools, data partitions | The lock owner can both read and update data in the locked object. Only uncommitted read (UR) applications can access the locked object. |
| **Z (Super Exclusive)** | Table spaces, tables, data partitions, blocks | This lock is acquired on a table under certain conditions, such as when the table is altered or dropped, an index on the table is created or dropped, or for some types of table reorganization. No other concurrent application can read or update the table. |

# Factors that affect locking

Several factors affect the mode and granularity of database manager locks.

These factors include:

- The type of processing that the application performs
- The data access method
- The values of various configuration parameters

## Locks and types of application processing

For the purpose of determining lock attributes, application processing can be classified as one of the following types: read-only, intent to change, change, and cursor controlled.

- Read-only

  This processing type includes all SELECT statements that are intrinsically read-only, have an explicit FOR READ ONLY clause, or are ambiguous, but the query compiler assumes that they are read-only because of the BLOCKING option value that the **PREP** or **BIND** command specifies. This type requires only share locks (IS, NS, or S).

- Intent to change

  This processing type includes all SELECT statements that have a FOR UPDATE clause, a USE AND KEEP UPDATE LOCKS clause, a USE AND KEEP EXCLUSIVE LOCKS clause, or are ambiguous, but the query compiler assumes that change is intended. This type uses share and update locks (S, U, or X for rows; IX, S, U, or X for blocks; and IX, U, or X for tables).

- Change

  This processing type includes UPDATE, INSERT, and DELETE statements, but not UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF. This type requires exclusive locks (IX or X).

- Cursor controlled

  This processing type includes UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF. This type requires exclusive locks (IX or X).

A statement that inserts, updates, or deletes data in a target table, based on the result from a subselect statement, does two types of processing. The rules for read-only processing determine the locks for the tables that return data in the subselect statement. The rules for change processing determine the locks for the target table.

## Locks and data-access methods

An *access plan* is the method that the optimizer selects to retrieve data from a specific table. The access plan can have a significant effect on lock modes.

If an index scan is used to locate a specific row, the optimizer will usually choose row-level locking (IS) for the table. For example, if the EMPLOYEE table has an index on employee number (EMPNO), access through that index might be used to select information for a single employee:

```
select * from employee
  where empno = '000310'
```

If an index is not used, the entire table must be scanned in sequence to find the required rows, and the optimizer will likely choose a single table-level lock (S). For example, if there is no index on the column SEX, a table scan might be used to select all male employees, as follows:

```
select * from employee
  where sex = 'M'
```

**Note:** Cursor-controlled processing uses the lock mode of the underlying cursor until the application finds a row to update or delete. For this type of processing, no matter what the lock mode of the cursor might be, an exclusive lock is always obtained to perform the update or delete operation.

Locking in range-clustered tables works slightly differently from standard key locking. When accessing a range of rows in a range-clustered table, all rows in the range are locked, even when some of those rows are empty. In standard key locking, only rows with existing data are locked.

Deferred access to data pages implies that access to a row occurs in two steps, which results in more complex locking scenarios. The timing of lock acquisition and the persistence of locks depend on the isolation level. Because the repeatable read (RR) isolation level retains all locks until the end of a transaction, the locks acquired in the first step are held, and there is no need to acquire further locks during the second step. For the read stability (RS) and cursor stability (CS) isolation levels, locks must be acquired during the second step. To maximize concurrency, locks are not acquired during the first step, and the reapplication of all predicates ensures that only qualifying rows are returned.

# Lock type compatibility

Lock compatibility becomes an issue when one application holds a lock on an object and another application requests a lock on the same object. When the two lock modes are compatible, the request for a second lock on the object can be granted.

If the lock mode of the requested lock is not compatible with the lock that is already held, the lock request cannot be granted. Instead, the request must wait until the first application releases its lock, and all other existing incompatible locks are released.

Table 79 shows which lock types are compatible (indicated by a **yes**) and which types are not (indicated by a **no**). Note that a timeout can occur when a requestor is waiting for a lock.

*Table 79. Lock Type Compatibility*

| | State of Held Resource | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **State Being Requested** | **None** | **IN** | **IS** | **NS** | **S** | **IX** | **SIX** | **U** | **X** | **Z** | **NW** |
| **None** | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes |
| **IN (Intent None)** | yes | yes | yes | yes | yes | yes | yes | yes | yes | no | yes |
| **IS (Intent Share)** | yes | yes | yes | yes | yes | yes | yes | yes | no | no | no |
| **NS (Scan Share)** | yes | yes | yes | yes | yes | no | no | yes | no | no | yes |
| **S (Share)** | yes | yes | yes | yes | yes | no | no | yes | no | no | no |
| **IX (Intent Exclusive)** | yes | yes | yes | no | no | yes | no | no | no | no | no |
| **SIX (Share with Intent Exclusive)** | yes | yes | yes | no | no | no | no | no | no | no | no |
| **U (Update)** | yes | yes | yes | yes | yes | no | no | no | no | no | no |
| **X (Exclusive)** | yes | yes | no | no | no | no | no | no | no | no | no |
| **Z (Super Exclusive)** | yes | no | no | no | no | no | no | no | no | no | no |
| **NW (Next Key Weak Exclusive)** | yes | yes | no | yes | no | no | no | no | no | no | no |

## Next-key locking

During insertion of a key into an index, the row that corresponds to the key that will follow the new key in the index is locked only if that row is currently locked by a repeatable read (RR) index scan. When this occurs, insertion of the new index key is deferred until the transaction that performed the RR scan completes.

The lock mode that is used for the next-key lock is NW (next key weak exclusive). This next-key lock is released before key insertion occurs; that is, before a row is inserted into the table.

Key insertion also occurs when updates to a row result in a change to the value of the index key for that row, because the original key value is marked deleted and the new key value is inserted into the index. For updates that affect only the include columns of an index, the key can be updated in place, and no next-key locking occurs.

During RR scans, the row that corresponds to the key that follows the end of the scan range is locked in S mode. If no keys follow the end of the scan range, an end-of-table lock is acquired to lock the end of the index. In the case of partitioned indexes for partitioned tables, locks are acquired to lock the end of each index partition, instead of just one lock for the end of the index. If the key that follows the end of the scan range is marked deleted, one of the following actions occurs:

- The scan continues to lock the corresponding rows until it finds a key that is not marked deleted
- The scan locks the corresponding row for that key
- The scan locks the end of the index

## Lock modes and access plans for standard tables

The type of lock that a standard table obtains depends on the isolation level that is in effect and on the data access plan that is being used.

The following tables show the types of locks that are obtained for standard tables under each isolation level for different access plans. Each entry has two parts: the table lock and the row lock. A hyphen indicates that a particular lock granularity is not available.

Tables 7-12 show the types of locks that are obtained when the reading of data pages is deferred to allow the list of rows to be further qualified using multiple indexes, or sorted for efficient prefetching.

- Table 1. Lock Modes for Table Scans with No Predicates
- Table 2. Lock Modes for Table Scans with Predicates
- Table 3. Lock Modes for RID Index Scans with No Predicates
- Table 4. Lock Modes for RID Index Scans with a Single Qualifying Row
- Table 5. Lock Modes for RID Index Scans with Start and Stop Predicates Only
- Table 6. Lock Modes for RID Index Scans with Index and Other Predicates (sargs, resids) Only
- Table 7. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates
- Table 8. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates
- Table 9. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)
- Table 10. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)
- Table 11. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only
- Table 12. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only

**Note:**

1. Block-level locks are also available for multidimensional clustering (MDC) and insert time clustering (ITC) tables.
2. Lock modes can be changed explicitly with the *lock-request-clause* of a SELECT statement.

*Table 80. Lock Modes for Table Scans with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/- | U/- | SIX/X | X/- | X/- |
| RS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| UR | IN/- | IX/U | IX/X | IX/X | IX/X |

*Table 81. Lock Modes for Table Scans with Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/- | U/- | SIX/X | U/- | SIX/X |
| RS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| UR | IN/- | IX/U | IX/X | IX/U | IX/X |

**Note:** Under the UR isolation level, if there are predicates on include columns in the index, the isolation level is upgraded to CS and the locks are upgraded to an IS table lock or NS row locks.

*Table 82. Lock Modes for RID Index Scans with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/- | IX/S | IX/X | X/- | X/- |
| RS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| UR | IN/- | IX/U | IX/X | IX/X | IX/X |

*Table 83. Lock Modes for RID Index Scans with a Single Qualifying Row*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S | IX/U | IX/X | IX/X | IX/X |
| RS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| UR | IN/- | IX/U | IX/X | IX/X | IX/X |

*Table 84. Lock Modes for RID Index Scans with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S | IX/S | IX/X | IX/X | IX/X |
| RS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| UR | IN/- | IX/U | IX/X | IX/X | IX/X |

*Table 85. Lock Modes for RID Index Scans with Index and Other Predicates (sargs, resids) Only*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S | IX/S | IX/X | IX/S | IX/X |
| RS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| UR | IN/- | IX/U | IX/X | IX/U | IX/X |

*Table 86. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S | IX/S | | X/- | |
| RS | IN/- | IN/- | | IN/- | |
| CS | IN/- | IN/- | | IN/- | |
| UR | IN/- | IN/- | | IN/- | |

*Table 87. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/- | IX/S | IX/X | X/- | X/- |
| RS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/X | IX/X |
| UR | IN/- | IX/U | IX/X | IX/X | IX/X |

*Table 88. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S | IX/S | | IX/S | |
| RS | IN/- | IN/- | | IN/- | |
| CS | IN/- | IN/- | | IN/- | |
| UR | IN/- | IN/- | | IN/- | |

*Table 89. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/- | IX/S | IX/X | IX/S | IX/X |
| RS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| UR | IN/- | IX/U | IX/X | IX/U | IX/X |

*Table 90. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S | IX/S | | IX/X | |
| RS | IN/- | IN/- | | IN/- | |
| CS | IN/- | IN/- | | IN/- | |
| UR | IN/- | IN/- | | IN/- | |

*Table 91. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operations | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/- | IX/S | IX/X | IX/X | IX/X |
| RS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| CS | IS/NS | IX/U | IX/X | IX/U | IX/X |
| UR | IS/- | IX/U | IX/X | IX/U | IX/X |

## Lock modes for MDC and ITC tables and RID index scans

The type of lock that a multidimensional clustering (MDC) or insert time clustering (ITC) table obtains during a table or RID index scan depends on the isolation level that is in effect and on the data access plan that is being used.

The following tables show the types of locks that are obtained for MDC and ITC tables under each isolation level for different access plans. Each entry has three parts: the table lock, the block lock, and the row lock. A hyphen indicates that a particular lock granularity is not available.

Tables 9-14 show the types of locks that are obtained for RID index scans when the reading of data pages is deferred. Under the UR isolation level, if there are predicates on include columns in the index, the isolation level is upgraded to CS and the locks are upgraded to an IS table lock, an IS block lock, or NS row locks.

- Table 1. Lock Modes for Table Scans with No Predicates
- Table 2. Lock Modes for Table Scans with Predicates on Dimension Columns Only
- Table 3. Lock Modes for Table Scans with Other Predicates (sargs, resids)
- Table 4. Lock Modes for RID Index Scans with No Predicates
- Table 5. Lock Modes for RID Index Scans with a Single Qualifying Row
- Table 6. Lock Modes for RID Index Scans with Start and Stop Predicates Only
- Table 7. Lock Modes for RID Index Scans with Index Predicates Only
- Table 8. Lock Modes for RID Index Scans with Other Predicates (sargs, resids)
- Table 9. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates
- Table 10. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates
- Table 11. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)
- Table 12. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)
- Table 13. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only
- Table 14. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only

**Note:** Lock modes can be changed explicitly with the *lock-request-clause* of a SELECT statement.

*Table 92. Lock Modes for Table Scans with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/-/- | U/-/- | SIX/IX/X | X/-/- | X/-/- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/U | IX/X/- | IX/I/- |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/X/- | IX/X/- |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/X/- | IX/X/- |

*Table 93. Lock Modes for Table Scans with Predicates on Dimension Columns Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/-/- | U/-/- | SIX/IX/X | U/-/- | SIX/X/- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/U/- | X/X/- |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/U/- | X/X/- |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/U/- | X/X/- |

*Table 94. Lock Modes for Table Scans with Other Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/-/- | U/-/- | SIX/IX/X | U/-/- | SIX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

*Table 95. Lock Modes for RID Index Scans with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/-/- | IX/IX/S | IX/IX/X | X/-/- | X/-/- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/X | X/X/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/X | X/X/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | X/X/X | X/X/X |

*Table 96. Lock Modes for RID Index Scans with a Single Qualifying Row*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/IS/S | IX/IX/U | IX/IX/X | X/X/X | X/X/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/X | X/X/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/X | X/X/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | X/X/X | X/X/X |

*Table 97. Lock Modes for RID Index Scans with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/IS/S | IX/IX/S | IX/IX/X | IX/IX/X | IX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/X | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/X | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/X | IX/IX/X |

*Table 98. Lock Modes for RID Index Scans with Index Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/S | IX/IX/S | IX/IX/X | IX/IX/S | IX/IX/X |

*Table 98. Lock Modes for RID Index Scans with Index Predicates Only (continued)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

*Table 99. Lock Modes for RID Index Scans with Other Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/S | IX/IX/S | IX/IX/X | IX/IX/S | IX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

*Table 100. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/S | IX/IX/S | | X/-/- | |
| RS | IN/IN/- | IN/IN/- | | IN/IN/- | |
| CS | IN/IN/- | IN/IN/- | | IN/IN/- | |
| UR | IN/IN/- | IN/IN/- | | IN/IN/- | |

*Table 101. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/- | IX/IX/S | IX/IX/X | X/-/- | X/-/- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/X | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/X | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/X | IX/IX/X |

*Table 102. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/- | IX/IX/S | | IX/IX/S | |
| RS | IN/IN/- | IN/IN/- | | IN/IN/- | |
| CS | IN/IN/- | IN/IN/- | | IN/IN/- | |
| UR | IN/IN/- | IN/IN/- | | IN/IN/- | |

*Table 103. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/- | IX/IX/S | IX/IX/X | IX/IX/S | IX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

*Table 104. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/IS/S | IX/IX/S | | IX/IX/X | |
| RS | IN/IN/- | IN/IN/- | | IN/IN/- | |
| CS | IN/IN/- | IN/IN/- | | IN/IN/- | |
| UR | IN/IN/- | IN/IN/- | | IN/IN/- | |

*Table 105. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/- | IX/IX/S | IX/IX/X | IX/IX/X | IX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IS/-/- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

## Lock modes for MDC block index scans

The type of lock that a multidimensional clustering (MDC) table obtains during a block index scan depends on the isolation level that is in effect and on the data access plan that is being used.

The following tables show the types of locks that are obtained for MDC tables under each isolation level for different access plans. Each entry has three parts: the table lock, the block lock, and the row lock. A hyphen indicates that a particular lock granularity is not available.

Tables 5-12 show the types of locks that are obtained for block index scans when the reading of data pages is deferred.

- Table 1. Lock Modes for Index Scans with No Predicates
- Table 2. Lock Modes for Index Scans with Predicates on Dimension Columns Only
- Table 3. Lock Modes for Index Scans with Start and Stop Predicates Only
- Table 4. Lock Modes for Index Scans with Predicates
- Table 5. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with No Predicates
- Table 6. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with No Predicates
- Table 7. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Predicates on Dimension Columns Only
- Table 8. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Predicates on Dimension Columns Only
- Table 9. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Start and Stop Predicates Only
- Table 10. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Start and Stop Predicates Only
- Table 11. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Other Predicates (sargs, resids)
- Table 12. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Other Predicates (sargs, resids)

**Note:** Lock modes can be changed explicitly with the *lock-request-clause* of a SELECT statement.

*Table 106. Lock Modes for Index Scans with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | S/--/-- | IX/IX/S | IX/IX/X | X/--/-- | X/--/-- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/-- | X/X/-- |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/-- | X/X/-- |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | X/X/-- | X/X/-- |

*Table 107. Lock Modes for Index Scans with Predicates on Dimension Columns Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/-/- | IX/IX/S | IX/IX/X | X/-/- | X/-/- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/X/- | IX/X/- |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/X/- | IX/X/- |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/X/- | IX/X/- |

*Table 108. Lock Modes for Index Scans with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/- | IX/IX/S | IX/IX/S | IX/IX/S | IX/IX/S |
| RS | IX/IX/S | IX/IX/U | IX/IX/X | IX/IX/- | IX/IX/- |
| CS | IX/IX/S | IX/IX/U | IX/IX/X | IX/IX/- | IX/IX/- |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/- | IX/IX/- |

*Table 109. Lock Modes for Index Scans with Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/- | IX/IX/S | IX/IX/X | IX/IX/S | IX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IN/IN/- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

*Table 110. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/-- | IX/IX/S | | X/--/-- | |
| RS | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |
| CS | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |
| UR | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |

*Table 111. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with No Predicates*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/-- | IX/IX/S | IX/IX/X | X/--/-- | X/--/-- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/-- | X/X/-- |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | X/X/-- | X/X/-- |
| UR | IN/IN/-- | IX/IX/U | IX/IX/X | X/X/-- | X/X/-- |

*Table 112. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Predicates on Dimension Columns Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/-- | IX/IX/-- | | IX/S/-- | |
| RS | IS/IS/NS | IX/--/-- | | IX/--/-- | |
| CS | IS/IS/NS | IX/--/-- | | IX/--/-- | |
| UR | IN/IN/-- | IX/--/-- | | IX/--/-- | |

*Table 113. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Predicates on Dimension Columns Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/-- | IX/IX/S | IX/IX/X | IX/S/-- | IX/X/-- |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/U/-- | IX/X/-- |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/U/-- | IX/X/-- |
| UR | IN/IN/-- | IX/IX/U | IX/IX/X | IX/U/-- | IX/X/-- |

*Table 114. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/-- | IX/IX/-- | | IX/X/-- | |
| RS | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |
| CS | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |
| UR | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |

*Table 115. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Start and Stop Predicates Only*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/-- | IX/IX/X | | IX/X/-- | |
| RS | IS/IS/NS | IN/IN/-- | | IN/IN/-- | |
| CS | IS/IS/NS | IN/IN/-- | | IN/IN/-- | |
| UR | IS/--/-- | IN/IN/-- | | IN/IN/-- | |

*Table 116. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Other Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IS/S/-- | IX/IX/-- | | IX/IX/-- | |
| RS | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |
| CS | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |
| UR | IN/IN/-- | IN/IN/-- | | IN/IN/-- | |

*Table 117. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Other Predicates (sargs, resids)*

| Isolation level | Read-only and ambiguous scans | Cursored operation | | Searched update or delete | |
|---|---|---|---|---|---|
| | | Scan | Where current of | Scan | Update or delete |
| RR | IN/IN/-- | IX/IX/S | IX/IX/X | IX/IX/S | IX/IX/X |
| RS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| CS | IS/IS/NS | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |
| UR | IN/IN/-- | IX/IX/U | IX/IX/X | IX/IX/U | IX/IX/X |

## Locking behavior on partitioned tables

In addition to an overall table lock, there is a lock for each data partition of a partitioned table.

This allows for finer granularity and increased concurrency compared to a nonpartitioned table. The data partition lock is identified in output from the **db2pd** command, event monitors, administrative views, and table functions.

When a table is accessed, a table lock is obtained first, and then data partition locks are obtained as required. Access methods and isolation levels might require the locking of data partitions that are not represented in the result set. After these data partition locks are acquired, they might be held as long as the table lock. For example, a cursor stability (CS) scan over an index might keep the locks on previously accessed data partitions to reduce the cost of reacquiring data partition locks later.

Data partition locks also carry the cost of ensuring access to table spaces. For nonpartitioned tables, table space access is handled by table locks. Data partition locking occurs even if there is an exclusive or share lock at the table level.

Finer granularity allows one transaction to have exclusive access to a specific data partition and to avoid row locking while other transactions are accessing other data partitions. This can be the result of the plan that is chosen for a mass update, or because of the escalation of locks to the data partition level. The table lock for many access methods is normally an intent lock, even if the data partitions are locked in share or exclusive mode. This allows for increased concurrency. However, if non-intent locks are required at the data partition level, and the plan indicates that all data partitions might be accessed, then a non-intent lock might be chosen at the table level to prevent data partition deadlocks between concurrent transactions.

## LOCK TABLE statements

For partitioned tables, the only lock acquired by the LOCK TABLE statement is a table-level lock. This prevents row locking by subsequent data manipulation language (DML) statements, and avoids deadlocks at the row, block, or data partition level. The IN EXCLUSIVE MODE option can be used to guarantee exclusive access when updating indexes, which is useful in limiting the growth of indexes during a large update.

## Effect of the LOCKSIZE TABLE option on the ALTER TABLE statement

The LOCKSIZE TABLE option ensures that a table is locked in share or exclusive mode with no intent locks. For a partitioned table, this locking strategy is applied to both the table lock and to data partition locks.

## Row- and block-level lock escalation

Row- and block-level locks in partitioned tables can be escalated to the data partition level. When this occurs, a table is more accessible to other transactions, even if a data partition is escalated to share, exclusive, or super exclusive mode, because other data partitions remain unaffected. The notification log entry for an escalation includes the impacted data partition and the name of the table.

Exclusive access to a nonpartitioned index cannot be ensured by lock escalation. For exclusive access, one of the following conditions must be true:

- The statement must use an exclusive table-level lock
- An explicit LOCK TABLE IN EXCLUSIVE MODE statement must be issued
- The table must have the LOCKSIZE TABLE attribute

In the case of partitioned indexes, exclusive access to an index partition is ensured by lock escalation of the data partition to an exclusive or super exclusive access mode.

## Interpreting lock information

The SNAPLOCK administrative view can help you to interpret lock information that is returned for a partitioned table. The following SNAPLOCK administrative view was captured during an offline index reorganization.

```
SELECT SUBSTR(TABNAME, 1, 15) TABNAME, TAB_FILE_ID, SUBSTR(TBSP_NAME, 1, 15) TBSP_NAME,
       DATA_PARTITION_ID, LOCK_OBJECT_TYPE, LOCK_MODE, LOCK_ESCALATION L_ESCALATION
  FROM SYSIBMADM.SNAPLOCK
  WHERE TABNAME like 'TP1' and LOCK_OBJECT_TYPE like 'TABLE_%'
  ORDER BY TABNAME, DATA_PARTITION_ID, LOCK_OBJECT_TYPE, TAB_FILE_ID, LOCK_MODE


TABNAME TAB_FILE_ID TBSP_NAME  DATA_PARTITION_ID LOCK_OBJECT_TYPE LOCK_MODE L_ESCALATION
------- ----------- ---------- ----------------- --------------- --------- ------------
TP1           32768 -                         -1 TABLE_LOCK      Z                    0
TP1               4 USERSPACE1                 0 TABLE_PART_LOCK Z                    0
TP1               5 USERSPACE1                 1 TABLE_PART_LOCK Z                    0
TP1               6 USERSPACE1                 2 TABLE_PART_LOCK Z                    0
TP1               7 USERSPACE1                 3 TABLE_PART_LOCK Z                    0
TP1               8 USERSPACE1                 4 TABLE_PART_LOCK Z                    0
TP1               9 USERSPACE1                 5 TABLE_PART_LOCK Z                    0
TP1              10 USERSPACE1                 6 TABLE_PART_LOCK Z                    0
TP1              11 USERSPACE1                 7 TABLE_PART_LOCK Z                    0
TP1              12 USERSPACE1                 8 TABLE_PART_LOCK Z                    0
TP1              13 USERSPACE1                 9 TABLE_PART_LOCK Z                    0
TP1              14 USERSPACE1                10 TABLE_PART_LOCK Z                    0
TP1              15 USERSPACE1                11 TABLE_PART_LOCK Z                    0
TP1               4 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               5 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               6 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               7 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               8 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               9 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              10 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              11 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              12 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              13 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              14 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              15 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              16 USERSPACE1                 - TABLE_LOCK      Z                    0

  26 record(s) selected.
```

In this example, a lock object of type TABLE_LOCK and a DATA_PARTITION_ID of -1 are used to control access to and concurrency on the partitioned table TP1. The lock objects of type TABLE_PART_LOCK are used to control most access to and concurrency on each data partition.

There are additional lock objects of type TABLE_LOCK captured in this output (TAB_FILE_ID 4 through 16) that do not have a value for DATA_PARTITION_ID. A lock of this type, where an object with a TAB_FILE_ID and a TBSP_NAME correspond to a data partition or index on the partitioned table, might be used to control concurrency with the online backup utility.

## Lock conversion

Changing the mode of a lock that is already held is called *lock conversion*.

Lock conversion occurs when a process accesses a data object on which it already holds a lock, and the access mode requires a more restrictive lock than the one already held. A process can hold only one lock on a data object at any given time, although it can request a lock on the same data object many times indirectly through a query.

Some lock modes apply only to tables, others only to rows, blocks, or data partitions. For rows or blocks, conversion usually occurs if an X lock is needed and an S or U lock is held.

IX and S locks are special cases with regard to lock conversion. Neither is considered to be more restrictive than the other, so if one of these locks is held and the other is required, the conversion results in a SIX (Share with Intent Exclusive) lock. All other conversions result in the requested lock mode becoming the held lock mode if the requested mode is more restrictive.

A dual conversion might also occur when a query updates a row. If the row is read through index access and locked as S, the table that contains the row has a covering intention lock. But if the lock type is IS instead of IX, and the row is subsequently changed, the table lock is converted to an IX and the row lock is converted to an X.

Lock conversion usually takes place implicitly as a query executes. The system monitor elements **lock_current_mode** and **lock_mode** can provide information about lock conversions occurring in your database.

## Lock escalation

A lock escalation occurs when, in the interest of reducing memory that is allocated to locks (lock space), numerous row-level locks are escalated into a single, memory-saving table lock. This situation, although automated and saves memory space devoted to locks, can reduce concurrency to an unacceptable level. You likely have a lock escalation problem if you are experiencing a higher than typical number of lock waits and the administration notification log entries indicate that lock escalations are occurring.

In general, to be able to objectively assess that your system is demonstrating abnormal behavior which can include processing delays and poor performance, you must have information that describes the typical behavior (baseline) of your system. A comparison can then be made between your observations of suspected abnormal behavior and the baseline. Collecting baseline data, by scheduling periodic operational monitoring tasks, is a key component of the troubleshooting process.

**Diagnosis**

Lock escalation from multiple row-level locks to a single table-level lock can occur for the following reasons:

- The total amount of memory consumed by many row-level locks held against a table exceeds the percentage of total memory allocated for storing locks
- The lock list runs out of space. The application that caused the lock list to be exhausted will have its locks forced through the lock escalation process, even though the application is not the holder of the most locks.

The threshold percentage of total memory allocated for storing locks, that has to be exceeded by an application for a lock escalation to occur, is defined by the **maxlocks** database configuration parameter and the allocated memory for locks is defined by the **locklist** database configuration parameter. In a well-configured database, lock escalation is rare. If lock escalation reduces concurrency to an unacceptable level, you can analyze the problem and decide on the best course of action.

Lock escalation is less of an issue, from the memory space perspective, if self tuning memory manager (STMM) is managing the memory for locks that is otherwise only allocated by the **locklist** database configuration

parameter. STMM will automatically adjust the memory space for locks if it ever runs out of free memory space.

**Indicative signs**

Look for the following indicative signs of lock escalations:

- Lock escalation message entries in the administration notification log

**What to monitor**

Due to the relatively transient nature of locking events, lock event data is most valuable if collected periodically over a period of time, so that the evolving picture can be better understood.

Check this monitoring element for indications that lock escalations might be a contributing factor in the SQL query performance slow down:

- **lock_escals**

If you have observed one or more of the indicative signs listed here, then you are likely experiencing a problem with lock escalations. Follow the link in the "What to do next" section to resolve this issue.

# Resolving lock escalation problems

After diagnosing a lock escalation problem, the next step is to attempt to resolve the issue resulting from the database manager automatically escalating locks from row level to table level. The guidelines provided here can help you to resolve the lock escalation problem you are experiencing and help you to prevent such future incidents.

## Before you begin

Confirm that you are experiencing a lock escalation problem by taking the necessary diagnostic steps for locking problems. For more details, see "Lock escalation" on page 437.

## About this task

The guidelines provided here can help you to resolve the lock escalation problem you are experiencing and help you to prevent such future incidents.

The objective is to minimize lock escalations, or eliminate them, if possible. A combination of good application design and database configuration for lock handling can minimize or eliminate lock escalations. Lock escalations can lead to reduced concurrency and potential lock timeouts, so addressing lock escalations is an important task. The **lock_escals** monitor element and messages written to the administration notification log can be used to identify and correct lock escalations.

First, ensure that lock escalation information is being recorded. Set the value of the **mon_lck_msg_lvl** database configuration parameter to 1. This is the default setting. When a lock escalation event occurs, information regarding the lock, workload, application, table, and error SQLCODEs are recorded. The query is also logged if it is a currently executing dynamic SQL statement.

## Procedure

Use the following steps to diagnose the cause of the unacceptable lock escalation problem and to apply a remedy:

1. Gather information from the administration notification log about all tables whose locks have been escalated and the applications involved. This log file includes the following information:
   - The number of locks currently held
   - The number of locks needed before lock escalation is completed
   - The table identifier and table name of each table being escalated
   - The number of non-table locks currently held
   - The new table-level lock to be acquired as part of the escalation. Usually, an S or X lock is acquired.
   - The internal return code that is associated with the acquisition of the new table-level lock
2. Use the administration notification log information about the applications involved in the lock escalations to decide how to resolve the escalation problems. Consider the following options:
   - Check and possibly adjust either the **maxlocks** or **locklist** database configuration parameters, or both. In a partitioned database system, make this change on all database partitions. The value of the **locklist** configuration parameter may be too small for your current workload. If multiple applications are experiencing lock escalation, this could be an indication that the lock list size needs to be increased. Growth in workloads or the addition of new applications could cause the lock list to be too small. If only one application is experiencing lock escalations, then adjusting the **maxlocks** configuration parameter could resolve this. However, you may want to consider increasing **locklist** at the same time you increase **maxlocks** - if one application is allowed to use more of the lock list, all the other applications could now exhaust the remaining locks available in the lock list and experience escalations.
   - You might want to consider the isolation level at which the application and the SQL statements are being run, for example RR, RS, CS, or UR. RR and RS isolation levels tend to cause more escalations because locks are held until a COMMIT is issued. CS and UR isolation levels do not hold locks until a COMMIT is issued, and therefore lock escalations are less likely. Use the lowest possible isolation level that can be tolerated by the application.
   - Increase the frequency of commits in the application, if business needs and the design of the application allow this. Increasing the frequency of commits reduces the number of locks that are held at any given time. This helps to prevent the application from reaching the **maxlocks** value, which triggers a lock escalation, and helps to prevent all the applications from exhausting the lock list.
   - You can modify the application to acquire table locks using the LOCK TABLE statement. This is a good strategy for tables where concurrent access by many applications and users is not critical; for example, when the application uses a permanent work table (for example, not a DGTT) that is uniquely named for this instance of the application. Acquiring table locks would be a good strategy in this case as it will reduce the number of locks being held by the application and increase the performance because row locks no longer need to be acquired and released on the rows that are accessed in the work table.

If the application does not have work tables and you cannot increase the values for **locklist** or **maxlocks** configuration parameters, then you can have the application acquire a table lock. However, care must be taken in choosing the table or tables to lock. Avoid tables that are accessed by many applications and users because locking these tables will lead to concurrency problems which can affect response time, and, in the worst case, can lead to applications experiencing lock timeouts.

### What to do next

Rerun the application or applications to ensure that the locking problem has been eliminated by checking the administration notification log for lock-related entries.

## Lock waits and timeouts

Lock timeout detection is a database manager feature that prevents applications from waiting indefinitely for a lock to be released.

For example, a transaction might be waiting for a lock that is held by another user's application, but the other user has left the workstation without allowing the application to commit the transaction, which would release the lock. To avoid stalling an application in such a case, set the **locktimeout** database configuration parameter to the maximum time that any application should have to wait to obtain a lock.

Setting this parameter helps to avoid global deadlocks, especially in distributed unit of work (DUOW) applications. If the time during which a lock request is pending is greater than the **locktimeout** value, an error is returned to the requesting application and its transaction is rolled back. For example, if APPL1 tries to acquire a lock that is already held by APPL2, APPL1 receives SQLCODE -911 (SQLSTATE 40001) with reason code 68 if the timeout period expires. The default value for **locktimeout** is -1, which means that lock timeout detection is disabled.

For table, row, data partition, and multidimensional clustering (MDC) block locks, an application can override the **locktimeout** value by changing the value of the CURRENT LOCK TIMEOUT special register.

To generate a report file about lock timeouts, set the **DB2_CAPTURE_LOCKTIMEOUT** registry variable to ON. The lock timeout report includes information about the key applications that were involved in lock contentions that resulted in lock timeouts, as well as details about the locks, such as lock name, lock type, row ID, table space ID, and table ID. Note that this variable is deprecated and might be removed in a future release because there are new methods to collect lock timeout events using the CREATE EVENT MONITOR FOR LOCKING statement.

To log more information about lock-request timeouts in the **db2diag** log files, set the value of the **diaglevel** database manager configuration parameter to 4. The logged information includes the name of the locked object, the lock mode, and the application that is holding the lock. The current dynamic SQL or XQuery statement or static package name might also be logged. A dynamic SQL or XQuery statement is logged only at **diaglevel** 4.

You can get additional information about lock waits and lock timeouts from the lock wait information system monitor elements, or from the **db.apps_waiting_locks** health indicator.

## Specifying a lock wait mode strategy

An session can specify a lock wait mode strategy, which is used when the session requires a lock that it cannot obtain immediately.

The strategy indicates whether the session will:

- Return an SQLCODE and SQLSTATE when it cannot obtain a lock
- Wait indefinitely for a lock
- Wait a specified amount of time for a lock
- Use the value of the **locktimeout** database configuration parameter when waiting for a lock

The lock wait mode strategy is specified through the SET CURRENT LOCK TIMEOUT statement, which changes the value of the CURRENT LOCK TIMEOUT special register. This special register specifies the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained.

Traditional locking approaches can result in applications blocking each other. This happens when one application must wait for another application to release its lock. Strategies to deal with the impact of such blocking usually provide a mechanism to specify the maximum acceptable duration of the block. That is the amount of time that an application will wait prior to returning without a lock. Previously, this was only possible at the database level by changing the value of the **locktimeout** database configuration parameter.

The value of **locktimeout** applies to all locks, but the lock types that are impacted by the lock wait mode strategy include row, table, index key, and multidimensional clustering (MDC) block locks.

# Deadlocks

A deadlock is created when two applications lock data that is needed by the other, resulting in a situation in which neither application can continue executing.

For example, in Figure 22 on page 442, there are two applications running concurrently: Application A and Application B. The first transaction for Application A is to update the first row in Table 1, and the second transaction is to update the second row in Table 2. Application B updates the second row in Table 2 first, and then the first row in Table 1. At time T1, Application A locks the first row in Table 1. At the same time, Application B locks the second row in Table 2. At time T2, Application A requests a lock on the second row in Table 2. However, at the same time, Application B tries to lock the first row in Table 1. Because Application A will not release its lock on the first row in Table 1 until it can complete an update to the second row in Table 2, and Application B will not release its lock on the second row in Table 2 until it can complete an update to the first row in Table 1, a deadlock occurs. The applications wait until one of them releases its lock on the data.

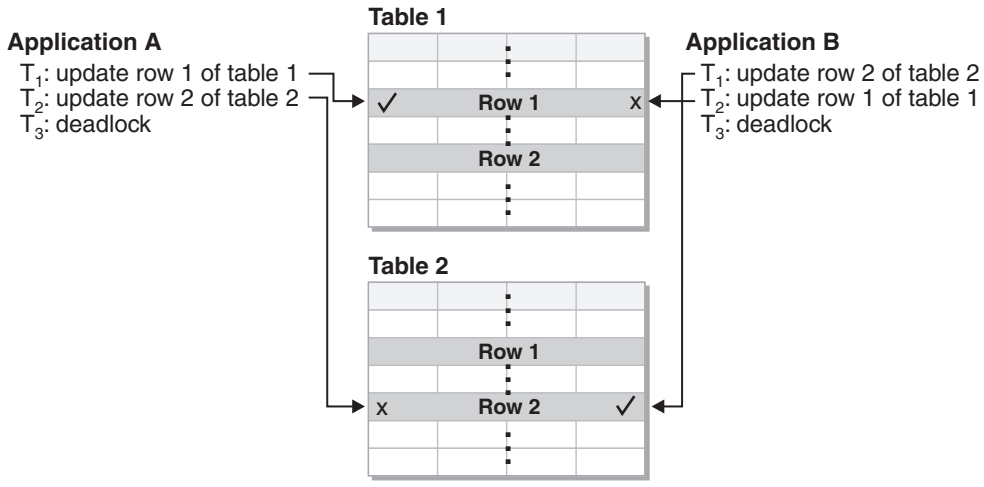



*Figure 22. Deadlock between applications*

Because applications do not voluntarily release locks on data that they need, a deadlock detector process is required to break deadlocks. The deadlock detector monitors information about agents that are waiting on locks, and awakens at intervals that are specified by the **dlchktime** database configuration parameter.

If it finds a deadlock, the deadlock detector arbitrarily selects one deadlocked process as the *victim process* to roll back. The victim process is awakened, and returns SQLCODE -911 (SQLSTATE 40001), with reason code 2, to the calling application. The database manager rolls back uncommitted transactions from the selected process automatically. When the rollback operation is complete, locks that belonged to the victim process are released, and the other processes involved in the deadlock can continue.

To ensure good performance, select an appropriate value for **dlchktime**. An interval that is too short causes unnecessary overhead, and an interval that is too long allows deadlocks to linger.

In a partitioned database environment, the value of **dlchktime** is applied only at the catalog database partition. If a large number of deadlocks are occurring, increase the value of **dlchktime** to account for lock waits and communication waits.

To avoid deadlocks when applications read data that they intend to subsequently update:

- Use the FOR UPDATE clause when performing a select operation. This clause ensures that a U lock is set when a process attempts to read data, and it does not allow row blocking.
- Use the WITH RR or WITH RS and USE AND KEEP UPDATE LOCKS clauses in queries. These clauses ensure that a U lock is set when a process attempts to read data, and they allow row blocking.

In a federated system, the data that is requested by an application might not be available because of a deadlock at the data source. When this happens, the DB2 server relies on the deadlock handling facilities at the data source. If deadlocks occur across more than one data source, the DB2 server relies on data source timeout mechanisms to break the deadlocks.

To log more information about deadlocks, set the value of the **diaglevel** database manager configuration parameter to 4. The logged information includes the name of the locked object, the lock mode, and the application that is holding the lock. The current dynamic SQL and XQuery statement or static package name might also be logged.

# Part 6. Appendixes

# Appendix A. Overview of the DB2 technical information

DB2 technical information is available in multiple formats that can be accessed in multiple ways.

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command-line help
  - Command help
  - Message help

**Note:** The DB2 Information Center topics are updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

### Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

## DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss. English and translated DB2 Version 10.1 manuals in PDF format can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg27009474.

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

**Note:** The *DB2 Information Center* is updated more frequently than either the PDF or the hard-copy books.

*Table 118. DB2 technical information*

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *Administrative API Reference* | SC27-3864-00 | Yes | April, 2012 |
| *Administrative Routines and Views* | SC27-3865-00 | No | January, 2013 |
| *Call Level Interface Guide and Reference Volume 1* | SC27-3866-00 | Yes | January, 2013 |
| *Call Level Interface Guide and Reference Volume 2* | SC27-3867-00 | Yes | January, 2013 |
| *Command Reference* | SC27-3868-00 | Yes | January, 2013 |
| *Database Administration Concepts and Configuration Reference* | SC27-3871-00 | Yes | January, 2013 |
| *Data Movement Utilities Guide and Reference* | SC27-3869-00 | Yes | January, 2013 |
| *Database Monitoring Guide and Reference* | SC27-3887-00 | Yes | January, 2013 |
| *Data Recovery and High Availability Guide and Reference* | SC27-3870-00 | Yes | January, 2013 |
| *Database Security Guide* | SC27-3872-00 | Yes | January, 2013 |
| *DB2 Workload Management Guide and Reference* | SC27-3891-00 | Yes | January, 2013 |
| *Developing ADO.NET and OLE DB Applications* | SC27-3873-00 | Yes | January, 2013 |
| *Developing Embedded SQL Applications* | SC27-3874-00 | Yes | January, 2013 |
| *Developing Java Applications* | SC27-3875-00 | Yes | January, 2013 |
| *Developing Perl, PHP, Python, and Ruby on Rails Applications* | SC27-3876-00 | No | April, 2012 |
| *Developing RDF Applications for IBM Data Servers* | | Yes | January, 2013 |
| *Developing User-defined Routines (SQL and External)* | SC27-3877-00 | Yes | January, 2013 |
| *Getting Started with Database Application Development* | GI13-2046-00 | Yes | January, 2013 |

Table 118. DB2 technical information (continued)

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *Getting Started with DB2 Installation and Administration on Linux and Windows* | GI13-2047-00 | Yes | April, 2012 |
| *Globalization Guide* | SC27-3878-00 | Yes | April, 2012 |
| *Installing DB2 Servers* | GC27-3884-00 | Yes | January, 2013 |
| *Installing IBM Data Server Clients* | GC27-3883-00 | No | April, 2012 |
| *Message Reference Volume 1* | SC27-3879-00 | No | January, 2013 |
| *Message Reference Volume 2* | SC27-3880-00 | No | January, 2013 |
| *Net Search Extender Administration and User's Guide* | SC27-3895-00 | No | January, 2013 |
| *Partitioning and Clustering Guide* | SC27-3882-00 | Yes | January, 2013 |
| *pureXML Guide* | SC27-3892-00 | Yes | January, 2013 |
| *Spatial Extender User's Guide and Reference* | SC27-3894-00 | No | April, 2012 |
| *SQL Procedural Languages: Application Enablement and Support* | SC27-3896-00 | Yes | January, 2013 |
| *SQL Reference Volume 1* | SC27-3885-00 | Yes | January, 2013 |
| *SQL Reference Volume 2* | SC27-3886-00 | Yes | January, 2013 |
| *Text Search Guide* | SC27-3888-00 | Yes | January, 2013 |
| *Troubleshooting and Tuning Database Performance* | SC27-3889-00 | Yes | January, 2013 |
| *Upgrading to DB2 Version 10.1* | SC27-3881-00 | Yes | January, 2013 |
| *What's New for DB2 Version 10.1* | SC27-3890-00 | Yes | January, 2013 |
| *XQuery Reference* | SC27-3893-00 | No | January, 2013 |

Table 119. DB2 Connect-specific technical information

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *DB2 Connect Installing and Configuring DB2 Connect Personal Edition* | SC27-3861-00 | Yes | April, 2012 |
| *DB2 Connect Installing and Configuring DB2 Connect Servers* | SC27-3862-00 | Yes | January, 2013 |
| *DB2 Connect User's Guide* | SC27-3863-00 | Yes | January, 2013 |

## Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

### Procedure

To start SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.
For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

## Accessing different versions of the DB2 Information Center

Documentation for other versions of DB2 products is found in separate information centers on ibm.com®.

### About this task

For DB2 Version 10.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v10r1.

For DB2 Version 9.8 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/.

For DB2 Version 9.7 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/.

For DB2 Version 9.5 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r5.

For DB2 Version 9.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the *DB2 Information Center* URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

## Updating the DB2 Information Center installed on your computer or intranet server

A locally installed DB2 Information Center must be updated periodically.

### Before you begin

A DB2 Version 10.1 Information Center must already be installed. For details, see the “Installing the DB2 Information Center using the DB2 Setup wizard” topic in *Installing DB2 Servers*. All prerequisites and restrictions that applied to installing the Information Center also apply to updating the Information Center.

## About this task

An existing DB2 Information Center can be updated automatically or manually:

- Automatic updates update existing Information Center features and languages. One benefit of automatic updates is that the Information Center is unavailable for a shorter time compared to during a manual update. In addition, automatic updates can be set to run as part of other batch jobs that run periodically.
- Manual updates can be used to update existing Information Center features and languages. Automatic updates reduce the downtime during the update process, however you must use the manual process when you want to add features or languages. For example, a local Information Center was originally installed with both English and French languages, and now you want to also install the German language; a manual update will install German, as well as, update the existing Information Center features and languages. However, a manual update requires you to manually stop, update, and restart the Information Center. The Information Center is unavailable during the entire update process. In the automatic update process the Information Center incurs an outage to restart the Information Center after the update only.

This topic details the process for automatic updates. For manual update instructions, see the "Manually updating the DB2 Information Center installed on your computer or intranet server" topic.

## Procedure

To automatically update the DB2 Information Center installed on your computer or intranet server:

1. On Linux operating systems,
   a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `/opt/ibm/db2ic/V10.1` directory.
   b. Navigate from the installation directory to the `doc/bin` directory.
   c. Run the `update-ic` script:
      ```
      update-ic
      ```
2. On Windows operating systems,
   a. Open a command window.
   b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `<Program Files>\IBM\DB2 Information Center\Version 10.1` directory, where `<Program Files>` represents the location of the Program Files directory.
   c. Navigate from the installation directory to the `doc\bin` directory.
   d. Run the `update-ic.bat` file:
      ```
      update-ic.bat
      ```

## Results

The DB2 Information Center restarts automatically. If updates were available, the Information Center displays the new and updated topics. If Information Center updates were not available, a message is added to the log. The log file is located in `doc\eclipse\configuration` directory. The log file name is a randomly generated number. For example, `1239053440785.log`.

# Manually updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

## About this task

Updating your locally installed *DB2 Information Center* manually requires that you:

1. Stop the *DB2 Information Center* on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. The Workstation version of the DB2 Information Center always runs in stand-alone mode. .
2. Use the Update feature to see what updates are available. If there are updates that you must install, you can use the Update feature to obtain and install them

   **Note:** If your environment requires installing the *DB2 Information Center* updates on a machine that is not connected to the internet, mirror the update site to a local file system by using a machine that is connected to the internet and has the *DB2 Information Center* installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.
   If update packages are available, use the Update feature to get the packages. However, the Update feature is only available in stand-alone mode.
3. Stop the stand-alone Information Center, and restart the *DB2 Information Center* on your computer.

**Note:** On Windows 2008, Windows Vista (and higher), the commands listed later in this section must be run as an administrator. To open a command prompt or graphical tool with full administrator privileges, right-click the shortcut and then select **Run as administrator**.

## Procedure

To update the *DB2 Information Center* installed on your computer or intranet server:

1. Stop the *DB2 Information Center*.
   - On Windows, click **Start** > **Control Panel** > **Administrative Tools** > **Services**. Then right-click **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:
     ```
     /etc/init.d/db2icdv10 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the *Program_Files*\IBM\DB2 Information Center\Version 10.1 directory, where *Program_Files* represents the location of the Program Files directory.
     c. Navigate from the installation directory to the doc\bin directory.
     d. Run the help_start.bat file:

```
help_start.bat
```

- On Linux:
  a. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `/opt/ibm/db2ic/V10.1` directory.
  b. Navigate from the installation directory to the `doc/bin` directory.
  c. Run the `help_start` script:

     ```
     help_start
     ```

The systems default Web browser opens to display the stand-alone Information Center.

3. Click the **Update** button ( ). (JavaScript must be enabled in your browser.) On the right panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the installation process, check that the selections you want to install, then click **Install Updates**.
5. After the installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center:
   - On Windows, navigate to the `doc\bin` directory within the installation directory, and run the `help_end.bat` file:

     ```
     help_end.bat
     ```

     **Note:** The `help_end` batch file contains the commands required to safely stop the processes that were started with the `help_start` batch file. Do not use `Ctrl-C` or any other method to stop `help_start.bat`.
   - On Linux, navigate to the `doc/bin` directory within the installation directory, and run the `help_end` script:

     ```
     help_end
     ```

     **Note:** The `help_end` script contains the commands required to safely stop the processes that were started with the `help_start` script. Do not use any other method to stop the `help_start` script.
7. Restart the *DB2 Information Center*.
   - On Windows, click **Start** > **Control Panel** > **Administrative Tools** > **Services**. Then right-click **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv10 start
     ```

### Results

The updated *DB2 Information Center* displays the new and updated topics.

## DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 database products. Lessons provide step-by-step instructions.

### Before you begin

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

### DB2 tutorials

To view the tutorial, click the title.

**"pureXML" in** *pureXML Guide*
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

## DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

**DB2 documentation**
Troubleshooting information can be found in the *Troubleshooting and Tuning Database Performance* or the Database fundamentals section of the *DB2 Information Center*, which contains:

- Information about how to isolate and identify problems with DB2 diagnostic tools and utilities.
- Solutions to some of the most common problem.
- Advice to help solve other problems you might encounter with your DB2 database products.

**IBM Support Portal**
See the IBM Support Portal if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the IBM Support Portal at http://www.ibm.com/support/entry/portal/Overview/Software/Information_Management/DB2_for_Linux,_UNIX_and_Windows

## Terms and conditions

Permissions for the use of these publications are granted subject to the following terms and conditions.

**Applicability:** These terms and conditions are in addition to any terms of use for the IBM website.

**Personal use:** You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

**Commercial use:** You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

**Rights:** Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

**IBM Trademarks:** IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml

# Appendix B. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

**The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:**
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements, changes, or both in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to websites not owned by IBM are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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 that has been exchanged, should contact:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

Such information may be available, subject to appropriate terms and conditions, including, in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

## Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

# Index

## A



## B



## C

# D

# E



# F



# G

# H



# I



# J



# K



# L

# M



# N



# O

# P



# Q



# R

# S

# T

# U



# V

# W



# X



# Z

Printed in USA

SC27-4540-00

Spine information:

IBM DB2 10.1 for Linux, UNIX, and Windows

**Preparation Guide for DB2 10.1 Fundamentals Exam 610**