Ingres® 9.3

Database Administrator Guide

INGRES

ING-93-DBA-05

Contents

Chapter 13: Performing Backup and Recovery 347

Chapter 1: Introducing Database Administration

This section contains the following topics:

In This Guide

This guide provides Ingres® database administrators with information about creating, maintaining, backing up, and recovering databases.

Audience

This guide is primarily intended for database administrators. In some cases, however, the responsibilities of the database administrator and the system administrator overlap. Therefore, some of the tasks and responsibilities described in this guide require permissions typically given to the system administrator, but not necessarily given to database administrators. In these cases, you must work with your system administrator to carry out these responsibilities.

Database Administrator Responsibilities

In Ingres, anyone who creates a database becomes the database administrator (DBA) for that database. Furthermore, there is no limit on the number of DBAs that can exist at a site.

Note: Before you can create a database, you must have the createdb privilege.

The DBA has permission to do the following:

- Create and destroy databases
- Manage public database objects
- Maintain database and query performance
- Monitor locking to maximize concurrency
- Back up and recover the database
- Authorize databases to use alternate locations
- Manage user access to data through grants on tables, views, procedures, and other objects.

  For information on managing user access, see the *Security Guide*.

Database Administration Summary

The following table is a quick reference to the statements, commands, and utilities used to perform database administration tasks. Refer to the appropriate chapter for details. Most of these tasks can be also performed using the Visual Tools on Windows.

Task	**Statement, Command, or Utility**
Authorize user access (For details, see the *Security Guide*.)	CREATE USER ALTER USER DROP USER accessdb
Create and delete databases	createdb destroydb
Define locations and extend a database to use alternate locations	CREATE LOCATION ALTER LOCATION DROP LOCATION

Task	Statement, Command, or Utility	
	accessdb	
Change locations of or duplicate a database	relocatedb	

Create tables, views, and schemas	CREATE TABLE
	CREATE VIEW
	CREATE SCHEMA AUTHORIZATION	

Use supplementary table statements as needed	DECLARE GLOBAL TEMPORARY TABLE	
Copy data from tables into files	COPY INTO	
Populate tables with data	COPY FROM INSERT	
Unload and then copy data from one database to another	unloaddb copydb	
Change ownership of a database or database object	unloaddb copydb copyform copyrep	
Maintain database tables	verifydb	
Grant permissions for access to tables, views, and database procedures (For details, see the *Security Guide*.)	GRANT REVOKE	
Grant database privileges (For details, see the *Security Guide*.)	GRANT ON DATABASE REVOKE	
Create, alter, drop groups and roles (For details, see the *Security Guide*.)	CREATE GROUP ALTER GROUP DROP GROUP	CREATE ROLE ALTER ROLE DROP ROLE
Create, execute, and drop database procedures	CREATE PROCEDURE EXECUTE PROCEDURE DROP PROCEDURE	
Set the levels of security auditing and monitor the security audit log file	ENABLE SECURITY_AUDIT DISABLE SECURITY_AUDIT CREATE | DROP | HELP SECURITY_ALARM REGISTER TABLE	

Task	Statement, Command, or Utility
	REMOVE TABLE

Establish integrity constraints for data	ALTER TABLE...ADD CONSTRAINT
	CREATE RULE
	DROP RULE
	CREATE PROCEDURE

Define database events	CREATE DBEVENT and other DBEVENT statements

Select storage structures of tables and create secondary indexes	SET RESULT_STRUCTURE
	CREATE TABLE AS
	CREATE INDEX

Change storage structures and minimize overflow to maintain query performance	MODIFY usermod
Optimize database performance by collecting statistics for the query optimizer, rebuilding system catalog table indexes, and monitoring query execution plans	optimizedb sysmod statdump
Manage locking strategies implemented by application developers	SET LOCKMODE
Back up and recover the database	ckpdb auditdb rollforwarddb
Monitor and improve performance of database queries	Various statements, commands, and utilities

What You Need to Know

This guide assumes that you are familiar with the windowing system on the target platform of the installation, including terminology, navigational techniques, and working with standard items, such as menus and dialogs.

Query Language Used in this Guide

The industry standard query language, SQL, is used as the standard query language throughout this guide. Ingres is compliant with ISO Entry SQL-92. For details about the settings required to operate in compliance with ISO Entry SQL-92, see the *SQL Reference Guide*.

System-specific Text in this Guide

Generally, Ingres operates the same way on all systems. When necessary, however, this guide provides information specific to your operating system. For example:

UNIX: Information is specific to the UNIX environment.

VMS: Information is specific to the VMS environment.

Windows: Information is specific to the Windows environment.

When necessary for clarity, the symbol ⧗ is used to indicate the end of system-specific text.

For sections that pertain to one system only, the system is indicated in the section title.

Terminology Used in this Guide

This guide uses the following terminology:

- A *command* is an operation that you execute at the operating system level.
- A *statement* is an operation that you embed in a program or execute interactively from a terminal monitor.

Note: A statement can be written in Ingres 4GL, a host programming language (such as C), or a database query language (SQL or QUEL).

Syntax Conventions Used in this Guide

This guide uses the following conventions to describe command and statement syntax:

Convention	**Usage**
`Monospace`	Indicates keywords, symbols, or punctuation that you must enter as shown.
Italics	Represent a variable name for which you must supply a value.
[ ] (brackets)	Indicate an optional item.
{ } (braces)	Indicate an optional item that you can repeat as many times as appropriate.
| (vertical bar)	Separates items in a list and indicates that you must choose one item.

Chapter 2: Creating Databases and Using Alternate Locations

This section contains the following topics:

Types of Files in an Ingres Database

An Ingres database consists of several types of files:

Data

Data files contain the following:

- User tables and indexes created by an authorized user. These are referred to as user data files. For details, see the chapter "Managing Tables and Views."
- The system catalogs. These are dictionary tables that contain information about the database, such as descriptions of its tables, columns, and views. For a complete description of the system catalogs, see the appendix "System Catalogs."

Checkpoint

Checkpoint files contain a static copy of your entire database. A checkpoint file is created each time you take a checkpoint of your database.

Journal

Journal files contain dynamic records of changes made to the journaled tables in your database.

Dump

Dump files contain records of changes to the database that occurred during the checkpoint process. These files are used to recover a database that was checkpointed online.

For additional information about checkpoint, journal, and dump files, see the chapter "Performing Backup and Recovery."

Work

Work files are used for system work, such as sorting and creating temporary tables.

Working With Database Objects

A database object specifies the database name, database type, file locations, and other attributes.

You can perform the following basic operations on database objects:

- Create and alter database objects

  Note: You can create as many databases as your operating system allows.

- View existing database objects, including the detailed properties of each individual object
- Drop database objects

You can accomplish these tasks using the createdb, catalogdb, relocatedb, infodb, and destroydb system commands. For details, see the *Command Reference Guide*.

In VDBA, use the Databases branch in the Database Object Manager window.

Createdb Privilege

The createdb privilege is required to create a database. This privilege is required to use the createdb system command (or to use the equivalent operation in VDBA). This subject privilege is granted by default to the system administrator, who in turn can grant it to other users, such as database administrators.

How a Database Is Created

When you create a database the following occurs:

- The system catalogs in the master database (iidbdb) are updated.
- A new subdirectory is created, with the name of the database, under the database location for the database. Later, similar subdirectories are created under the work, journal, dump, and checkpoint locations for the database, as needed.
- The configuration file (aaaaaaaa.cnf) and the core system catalogs (aaaaaaa*x*.t00, *x*=b through e) are copied to the new database directory.
- The DBMS system catalogs for the new database are created and modified.
- The standard catalog interface is created.
- The user interface system catalogs (restricted by any -f flag options) are created.
- If creating a distributed database, Ingres Star system catalogs for the database are initialized and modified.
- Select permission for the system catalogs is granted to public.

Extend and Unextend a Database

You can extend a database to use additional data and work locations. Locations must exist prior to this operation and must be specified with a Usage Type of database or work.

To extend a database, use either the extenddb command or the Alter Database dialog in VDBA. For details on the extenddb command, see the *Command Reference Guide*.

After extending to another data location, create new tables and indexes in the extended location, and modify existing tables and indexes to use the extended location. For details on creating and moving tables, see the chapter "Managing Tables and Views."

When you extend to a new work location, the system spreads the workload between the initial location (specified at create time) and the extensions.

Unextending a database reverses the extend operation and deletes the entry from the configuration files so the location can be used again.

Note: After unextending a database location, you should checkpoint the database. Previous checkpoints cannot be used because they reference a location that is no longer accessible to the database.

To unextend a database, use either the unextenddb command or the Alter Database dialog in VDBA. For additional information, see the *Command Reference Guide* and online help, respectively.

Relocate Database Files

You can relocate journal, checkpoint, and dump work files for an existing database. Locations must exist prior to this operation and must be specified with an appropriate Usage Type (that is, journal, checkpoint, or dump, depending on the type of file you want to relocate). When you relocate checkpoint, journal, or dump files, the existing files are moved to the new location and any new files are created there.

To relocate database files, use the Alter Database dialog in VDBA. For more information, see the online help topic Altering a Database.

You can also accomplish this task using the relocatedb system command. For more information, see the *Command Reference Guide*.

How a Database Is Dropped

When you drop a database, the following occurs:

- The database, checkpoint, journal, dump, and work directories for the database are deleted.
- All traces of the database are removed from the master database (iidbdb).
- The Application-By-Forms object file directories for any applications associated with the database (but not the source code directories) are deleted.

Caution! Do not set ING_ABFDIR to be your default login directory or other directory that contains your own files. Your files can be inadvertently destroyed if a destroydb *dbname* command is issued and *dbname* is the same name as the ING_ABFDIR directory.

Locations and Areas

Each database file type (data, checkpoint, journal, and so on) is associated with a location, which maps to a specific disk volume or directory, called an *area*.

Default Locations

During installation, default storage locations and underlying areas are established for each type of database file.

When you create a database, the Ingres default locations are assumed unless you specify alternate locations.

The following table shows the default locations and the Ingres environment variables that identify the areas to which the locations are mapped:

File Type	Location Name	Area
Data	ii_database	II_DATABASE
Checkpoint	ii_checkpoint	II_CHECKPOINT
Journal	ii_journal	II_JOURNAL
Dump	ii_dump	II_DUMP
Work	ii_work	II_WORK

In each case, the Ingres environment variable points to a specific disk volume or directory that has a particular structure, which is shown in the following tables.

Windows:

File Type	Structure
Data	ingres\data\default
Checkpoint	ingres\ckp\default
Journal	ingres\jnl\default
Dump	ingres\dmp\default
Work	ingres\work\default

For example, using the default location for data files causes them to be stored in the *ii_database*\ingres\data\default directory, where *ii_database* is the value displayed by the ingprenv command for the II_DATABASE environment variable.

UNIX:

File Type	Structure
Data	ingres/data/default
Checkpoint	ingres/ckp/default
Journal	ingres/jnl/default
Dump	ingres/dmp/default
Work	ingres/work/default

For example, using the default location for dump files causes them to be stored in the *ii_dump*/ingres/dmp/default directory, where *ii_dump* is the value displayed by the ingprenv command for the II_DUMP environment variable.

VMS:

File Type	Structure
Data	[INGRES.DATA]
Checkpoint	[INGRES.CKP]
Journal	[INGRES.JNL]
Dump	[INGRES.DMP]

File Type	Structure
Work	[INGRES.WORK]

For example, using the default location for work files causes them to be stored in the *ii_work*:[INGRES.WORK] directory, where *ii_work* is the value displayed by the show logical command for the II_WORK environment variable.

Alternate Locations

You can use alternate locations for a new database, but first you must create the area (directory structure) where the files will be stored, and then define their location.

You create a location's area using the facilities of the host operating system.

Each area must have a specific subdirectory structure, depending on the file types with which it is associated. This structure parallels that of the corresponding default location area, as summarized in Default Locations (see page 25).

Create an Area in Windows

An area must be created before you can define an alternate location for a new database.

Note: If you use the extenddb command with the –a*area_dir* flag, the area is created for you. You do not have to create the directory path below the ingres root directory.

To create an area in Windows, follow these steps:

1. Change location to the drive and directory where you create the new directory structure. For example, to create the new directory structure on the D: drive under the \otherplace directory, issue the following commands at the command prompt:

   ```
   D:
   cd \otherplace
   ```

2. Create a new subdirectory. For example, to make a subdirectory named new_area, issue the following command at the command prompt:

   ```
   mkdir new_area
   ```

3. Create subdirectories for the types of database files that use the new area. For example, to create a subdirectory for data files in new_area, issue these commands at the command prompt:

   ```
   mkdir new_area\ingres
   mkdir new_area\ingres\data
   mkdir new_area\ingres\data\default
   ```

   To make subdirectories for checkpoint, journal, dump, or work files, substitute ckp, jnl, dmp, or work for data when issuing these commands.

In these steps, you created the area D:\otherplace\new_area, which you can now specify as the Area when defining a new location using the Create Location dialog in VDBA. The subdirectories you created in Step 3 determine which Usage Types you can select in this same dialog (and in the Alter Location dialog). For example, creating ingres\data\default allows you to enable Database as a Usage Type, and creating ingres\work\default allows you to enable Work as a Usage Type.

Create an Area in UNIX

An area must be created before you can define an alternate location for a new database.

Note: If you use the extenddb command with the –a*area_dir* flag, the area is created for you. You do not have to create the directory path below the ingres root directory.

To create an area in UNIX, follow these steps:

1. Log in as the installation owner.

   By using this account, this user becomes the owner of the subdirectories created in this procedure.

2. Change location to the directory where you create the new directory structure. For example, to create the new directory structure in the otherplace directory, issue the following command at the operating system prompt:

   ```
   cd /otherplace
   ```

   The installation owner account must be able to create a directory below this directory; this means permissions set to at least 755. If this number needs to be changed, see your system administrator. Top-level directories are usually managed by root.

3. Create a new subdirectory. For example, to make a subdirectory named new_area, issue the following command at the operating system prompt:

   ```
   mkdir new_area
   ```

4. Create subdirectories for the types of database files that use the new area. For example, to create a subdirectory for data files in new_area, issue these commands at the operating system prompt:

   ```
   mkdir new_area/ingres
   mkdir new_area/ingres/data
   mkdir new_area/ingres/data/default
   ```

   To make subdirectories for checkpoint, journal, dump, or work files, substitute ckp, jnl, dmp, or work for data when issuing the above commands.

5. Place the appropriate permissions on the new directories and subdirectories, as shown in the following example. Limit access to the data directory to the user account for the installation owner only:

   ```
   chmod 755 new_area
   chmod 755 new_area/ingres
   chmod 700 new_area/ingres/data
   chmod 777 new_area/ingres/data/default
   ```

   To place permissions on new directories for checkpoint, journal, dump, or work files, substitute ckp, jnl, dmp, or work for data when issuing the above commands.

In these steps, you created the area /otherplace/new_area, which you can now specify as the Area when defining a new location using the Create Location dialog in VDBA. The subdirectories you created in Step 4 determine which Usage Types you can select in this same dialog (and in the Alter Location dialog). For example, creating ingres/data/default allows you to enable Database as a Usage Type, and creating ingres/work/default allows you to enable Work as a Usage Type.

Raw Area in UNIX

A raw area contains data from a single database only.

A raw location can be assigned a usage of database only, cannot be used as the root location of a database, and can contain data for one table only.

The maximum size of a table is bound by the smallest raw location to which it is assigned.

A raw area can contain many locations; each location can contain the data for one table. A raw location is the equivalent of a cooked database file, which contains data of one table only.

To set up a raw area file, use the mkrawarea utility. For more information, see the *Command Reference Guide*.

How to Change from Raw to Cooked (Non-raw) Transaction Log

If your installation uses a raw transaction log file and you want to change to a cooked transaction log file, follow this process:

1. Destroy the existing transaction logs, including dual logs if present.
2. Define the locations to be used for the new transaction logs.
3. Create the new transaction logs.
4. Test the new transaction logs by restarting Ingres.

Create an Area in VMS

An area must be created before you can define an alternate location for a new database.

To create an area in VMS, follow these steps:

1. Log into the VMS system account.
2. Create the top level [INGRES] directory on the new device with the protection mask set to equal (S:RWE,O:RWE,G,W:RE) and ownership set to [INGRES] by executing the following command at the operating system prompt:

   ```
   CREATE/DIR device:[INGRES]/OWNER_UIC=[INGRES] -
   /PROT=(S:RWE,O:RWE,G,W:RE)
   ```

   Substitute the name of the new device for device in the command. Also, do not set the protections any more restrictive than recommended here, because doing so can result in errors later.
3. Make sure the master file directory [000000] on the new device has at least W:E protection by executing the following command at the operating system prompt:

   ```
   DIR/PROT device:[0,0]000000.dir
   ```

   If the protection is incorrect (for example, the WORLD has no access), correct this with the following command:

   ```
   SET FILE/PROT=(S:RWE, O:RWE, G, W:E) -
   device:[0,0]000000.dir
   ```
4. Define a logical name for the new area at the system level:

   ```
   DEFINE/SYSTEM/EXEC/TRANS=CONCEALED -
    logical_name device
   ```

   Substitute the name of the new area for logical_name. This is useful if you ever reconfigure your system or move data between systems, because it is much easier to redefine one logical than to re-point all references to a device.

   For example, the following command defines a new altarea1 for device DUA1 at the indicated subdirectory:

   ```
   DEFINE/SYSTEM/EXEC/trans=concealed -
   altarea1 dua1:[MYDIRECTORY.SUBDIRECTORY.]
   @II_SYSTEM:[INGRES.UTILITY]INGDEFDEV.COM
   ```
5. The definition in Step 4 lasts until the next system boot. Add the same DEFINE statement to SYS$MANAGER:SYSTARTUP_V5.COM or II_SYSTEM:[INGRES]IISTARTUP1.COM so that it is executed on future boots.
6. Exit the VMS system account.
7. Log in to the system administrator's account.

8. Create the subdirectories and set the appropriate protections on these directories by executing the INGDEFDEV command procedure at the operating system prompt:

9. When INGDEFDEV prompts you, provide the device name and the file type (data, journal, checkpoint, dump, or work) that resides in this area. Because you can specify only one file type each time you run INGDEFDEV, you must run INGDEFDEV once for each file type and device name pairing.

   Depending on the type of file that resides in this area, INGDEFDEV creates one of the following directories, where device is the name of the new device from Step 2:

   -device:[INGRES.DATA] (for data files)
   -device:[INGRES.CKP] (for checkpoint files)
   -device:[INGRES.JNL] (for journal files)
   -device:[INGRES.DMP] (for dump files)
   -device:[INGRES.WORK] (for work files)

In these steps, you created an area corresponding to the *logical_name* identified in Step 4, which you can now specify as the area name when defining a new location using the create location statement. The directories created by INGDEFDEV in Step 9 determine which usage types you can specify for both the create location and alter location statements. For example, creating [INGRES.DATA] allows you to specify usage = database, and creating [INGRES.WORK] allows you to specify usage = work.

Working with Locations

After you have created an area for a location, you must than create the location.

A location object specifies the location name, associated area, and the types of files that reside in the location.

You can perform the following basic operations on location objects:

- Create and alter location objects
- View existing location objects, including the detailed properties of each individual object
- Drop location objects

In SQL, you can manage locations using the CREATE LOCATION, ALTER LOCATION, and DROP LOCATION statements. For details, see the *SQL Reference Guide*.

In VDBA, use the Locations branch in the Database Object Manager window. For detailed steps, see online help for VDBA.

To work with locations, you need the maintain_locations privilege. This subject privilege is granted by default to the system administrator, who in turn can grant it to other users, such as database administrators, who need to manage locations. For more information on subject privileges, see the *Security Guide*.

Guidelines for Using Locations

After you have set up the underlying area and mapped it to a location by creating a location object, use the new location as summarized below:

- When you create a new database, specify the location for the database's data dump, checkpoint, journal, and work files.
- Extend a database to include the new location for its data and work files.
- After extending a database to use an alternate location designated for data files, move existing user data files (that is, user tables and indexes, but not the system catalogs) to it, and place new user data files in it. For more information, see the chapter "Managing Tables and Views."

- The following file types can use only a single location (that is, these file types are not affected when you extend a database):
  - Checkpoint files
  - Journal files
  - Dump files
- The initial location of the following file types is determined when you create a database, but you can move each to a new location (see page 24) if the need arises:
  - Checkpoint files
  - Journal files
  - Dump files
- Store the data, checkpoint, journal, dump, and work files for a database in the same locations or in different locations.
  - If the default locations are used when you create the database, all these files are stored in the same area.
  - We strongly recommend that you store data files on a different disk from those used to store checkpoints, journals, and dumps. Doing so helps to protect your data in the event of disk failure and to maximize disk drive performance.

The following table summarizes some of these guidelines:

File Type	**Extend to Use Multiple Locations?**	**Change Locations?**
Data	Yes	Yes (user tables and indexes) No (system catalogs)
Checkpoint	No	Yes
Journal	No	Yes
Dump	No	Yes
Work	Yes	No

Work Locations

All databases use work files for sorting, which can occur when queries are executed (SELECT or FETCH statements with ORDER BY clauses) or when tables are restructured (for example, using the MODIFY statement or equivalent operation in VDBA). While small sorts are performed in memory, larger sorts use temporary sort files. Depending on the size of the tables involved in the sort, the temporary disk space requirements can be large (see page 408).

Initial and Extended Work Locations

You specify the initial, or primary, location (or use the default location) for work files when you create a database. The area mapped to this location is used for all work files.

To use additional locations, extend a database. When you extend a database in this manner, sort space can be spread among multiple work locations.

Note: We recommend that you put work locations on scratch disks so that sorting activity does not contend with other database I/O and data disks do not become excessively fragmented.

Classification of Extended Work Locations

When extending a database to use additional work locations, classify them as follows:

- *Work* (also known as defaultable) locations are used for all user sorts on a database.
- *Auxiliary* locations are not used unless explicitly requested by a SET WORK LOCATIONS statement.

After a database has been extended to use an additional work location, you can subsequently modify the work area's classification using the Alter Database dialog in VDBA.

Work Locations for a Session

A session automatically uses all defaultable work locations to which the database has been extended (including the initial work location). In addition, the session can issue SET WORK LOCATIONS statements to specify auxiliary work locations to use. Using this statement, a session can dynamically add and drop work locations and replace the set of locations currently in use. The set work locations statements affect the current session only—their effects disappear when the session ends.

For more information on using set work locations, see the entry for the SET statement in the *SQL Reference Guide*.

Note: To list the set of work locations used in a given session, you can use a trace point, DM1440. For information on setting trace points, see the SET [NO]TRACE POINT statement description in the *SQL Reference Guide*.

Chapter 3: Managing Tables and Views

This section contains the following topics:

This chapter discusses how to manage tables, views, and schemas. It includes information on table limits, handling duplicate rows in tables, manipulating columns, modifying tables in various ways, and rules for updating views. It also discusses synonyms, temporary tables, and comments, which are features for manipulating table data and referencing tables.

Table Management

You can perform the following basic operations on tables:

- Create and alter table objects
- View existing table objects, including details such as the rows, columns, statistics, and other properties
- Modify table objects to change file locations
- Drop table objects

In VDBA, you use the Tables branch for a particular database in the Database Object Manager window.

In SQL, you can accomplish these tasks using the CREATE TABLE, ALTER TABLE, HELP TABLE, MODIFY TABLE, and DROP statements. For more information, see the *SQL Reference Guide*.

Tools for Creating a Table

You can create a table by issuing a CREATE TABLE statement from any of the following tools:

- A terminal monitor
- Interactive SQL
- An embedded SQL program
- Application-By-Forms and Ingres 4GL

For details on the CREATE TABLE statement, see the *SQL Reference Guide*.

You can also use the Tables utility to create tables. This utility lets you build and destroy tables, inspect their structure, and run queries and reports on their data.

UNIX: For a discussion of the Tables utility, see the *Character-based Querying and Reporting Tools User Guide*. ⧗

VMS: All users can create tables unless explicitly restricted from doing so using a nocreate_table database grant. ⧗

Table Ownership

The new table is owned by the user who creates it. The owner of the table is allowed to perform certain operations on the table that other users are not. For example, only the owner (or a user with the security privilege impersonating the owner) can alter or drop a table.

If other users need access to the table, the table owner can grant that permission using *table grants*. Table grants are *enabling* permissions—if no permission is granted, the default is to prohibit access.

Table Location

When you create a table, it is placed in the default location designated for the database's data files, unless you specify otherwise.

Requirements for Using an Alternate Location for a Table

Before using an alternate location for a table, the following requirements must be met:

- The location must exist and must be designated to hold data files
- The area to which the location name points must exist with correct permissions and ownership
- The directory indicated by the area must have the appropriate subdirectory structure with correct permissions
- The database must be extended to the location
- You must be the table owner (or a user with the security privilege impersonating the owner).

Alternate Location for a Table

To create a table in an alternate location, click Options in the Create Table dialog in VDBA. This opens the Options dialog, where you choose one or more alternate locations. For details, see online help.

If you specify only one location, the entire table is stored in that location. If you choose more than one location, the table spans multiple locations. For example, if you designate two locations, the table is extended over two alternate locations. As rows are added to the table, they are added to each area in alternate blocks.

The blocks are:

Windows: 16-page blocks (approximately 32 KB)

UNIX: 16-page blocks (approximately 32 KB)

VMS: 8 pages (32 disk blocks)

The table is considered out of space if the next receiving area in turn does not have a sufficient block.

Note: After creating a table, you can change its location, as described in Techniques for Moving a Table to a New Location (see page 58).

Enable or Disable Journaling

When you create a table, *journaling* can be enabled by default, depending on the setting of default_journaling in the Ingres DBMS Server class your session is attached to.

In VDBA, you can verify whether journaling is enabled or disabled by clicking Options in the Create Table dialog. This opens the Options dialog, which contains a Journaling check box. If it is checked, journaling is on (enabled); if is not checked, journaling is off (disabled).

By disabling the Journaling check box, you turn off journaling for an individual table, but use caution. For additional information about journaling and the ramifications of disabling journaling at the table level, see the chapter "Performing Backup and Recovery."

Duplicate Rows in Tables

A table contains duplicate rows when two or more rows are identical.

When you create a table, specify the handling of duplicate rows by clicking Options in the Create Table dialog. This opens the Options dialog, which contains a Duplicates check box. By default, duplicate rows are allowed in any new table that you create, which is indicated by the fact that the Duplicates check box is initially enabled. If you disable this check box, duplicate rows are not allowed in the table. If a user attempts to insert a duplicate row into a table where duplicate rows are not allowed, an error is generated.

Note: Duplicate rows are enforced only when the table has a keyed storage structure. For a description of storage structures, see the chapter "Choosing Storage Structures and Secondary Indexes."

Depending on whether duplicates are allowed, the following tasks are performed differently:

- Restructuring or relocating a table with the MODIFY statement or the equivalent operation in VDBA
- Adding new rows into a table with the INSERT statement
- Bulk loading a table with the COPY statement
- Revising existing rows in a table with the UPDATE command

Duplicate Rows When Adding New Rows or Modifying a Table

If a table was originally created to allow duplicates, the duplicate rows are preserved, even when the table is modified to another structure.

If a table allows duplicates, duplicate rows can always be inserted.

If a table does not allow duplicates:

- Duplicate rows can be added if the table uses a heap storage structure.
- Single row inserts (insert . . . values) are silently discarded if a duplicate row occurs on a keyed structure.
- Multiple row inserts (insert . . . select) generate an error if a duplicate row occurs on a keyed structure. The entire statement is rolled back.
- When a table is modified from a heap structure to a keyed structure, duplicates are eliminated.

Duplicate Rows When Bulk Copying Rows in a Table

If a table allows duplicates, duplicate rows can always be loaded.

If a table does not allow duplicates, duplicate rows:

- Can be loaded if the table uses a heap storage structure
- Are silently removed if the table has a keyed structure

Duplicate Rows in Updated Tables

If a table allows duplicates, rows can always be updated to duplicate other rows.

If a table does not allow duplicates:

- Rows can be updated to duplicate other rows if the table uses a heap storage structure.
- Rows cannot be updated to duplicate other rows if the table is a keyed structure. The update is rejected and an error is generated.

If you use the following "bulk increment" update in which the info column has values 1, 2, 3, and so on, every row in the table is updated:

```
update data set info = info+1;
```

If duplicates are not allowed, this update fails due to duplicate rows being created.

The new values for the first row are prepared, changing the info column value from 1 to 2. Before inserting the new values, a check is made to see if they violate any constraints. Because the new value, 2, duplicates the value in an existing row, thus violating the criterion stating that duplicates are not allowed, an error is generated and the update is rolled back.

To solve this problem, use either of the following techniques:

- Allow duplicates when creating the table
- Modify the table to use a heap storage structure before performing the update

Remove Duplicate Rows

In this example, assume the table from which you want to remove the duplicates is named has_dups. This example creates one table based upon the contents of another. For more information, see online help.

Follow these steps to remove duplicate rows:

1. Create a new table named temp.
2. Enable Create Table As Select in the Create Table dialog.
3. In the Select Statement edit control, enter:

   ```
   select distinct * from has_dups
   ```

4. Drop the has_dups table.
5. Create a new table named has_dups, using the Options dialog to disable the Duplicates check box.
6. Enable Create Table As Select in the Create Table dialog.
7. In the Select Statement edit control, enter:

   ```
   select * from temp
   ```

8. Drop the temp table.

   Note: If a table was originally created to allow duplicate rows, but you no longer want the table to allow duplicate rows, you must perform Steps 1-8 above. However, because duplicate row checking is only performed for tables having a keyed structure, you must also perform this additional step:

9. Modify the table to a keyed structure (hash, ISAM, or B-tree).

Page Size Specification

A *page* is a block of physical storage space.

When you create a table, specify its page size by clicking Options in the Create Table dialog. This opens the Options dialog, which contains a Page Size drop-down list box. If you need assistance, see online help for details.

The default page size is determined by the DBMS configuration parameter, default_ page_size. The corresponding buffer cache for the installation must also be configured with the page size you specify or you receive an error. For more information, see the chapter "Configuring Ingres" in the *System Administrator Guide*.

Note: After creating a table, if you later need to add or drop a column, the page size of the table must be larger than 2 KB. If you anticipate that a particular table needs to be altered in either of these ways, create the table using a larger page size or modify its storage structure before attempting to alter the table. For more information, see the chapter "Maintaining Storage Structures."

Data Type Conversion Functions for Default Values

When you create or alter a table, specify a default value for any column, which is used when no value is specified for the column. Instead of specifying a typical default value of zero or quoted spaces for a column, substitute a particular value as the default value for the new column. To do this, use the associated conversion function for the data type assigned to the new column.

The following table lists the data type and an example of its associated conversion function for creating a column:

Data Type	Conversion Function
char(1)	char(' ')
c1	c(' ')
varchar(7)	varchar(' ')
long varchar	long_varchar(' ')
nchar	nchar(' ')
nvarchar	nvarchar (' ')
text(7)	text(' ')
byte(binary)	byte(0)
long byte (binary)	long_byte(0)

Data Type	Conversion Function
byte varying (binary)	varbyte(0)
integer (integer4)	int4(0)
smallint (integer2)	int2(0)
integer1	int1(0)
float (float8)	float8(0)
float4	float4(null)
decimal	decimal(0)
ansidate	ansidate('') or ansidate(null)
time with time zone	time_with_tz(' ') or time_with_tz(null)
time without time zone	time_wo_tz(' ') or time_wo_tz(null)
time with local time zone	time_local(' ') or time_local(null)
timestamp with time zone	timestamp_with_tz(' ') or timestamp_with_tz(null)
timestamp without local time zone	timestamp_wo_tz(' ') or timestamp_wo_tz(null)
timestamp with local time zone	timestamp_local(' ') or timestamp_local(null)
interval day to second	interval_dtos(' ') or interval_dtos(null)
interval year to month	interval_ytom(' ') or interval_ytom(null)
ingresdate	ingresdate(' ') or ingresdate(null)
money	money(0)
object_key	object_key('01')
table_key	table_key('01')

If the new column is created with no conversion function, the defaults are:

- varchar for character strings
- float (float8) for floating point numbers
- Either smallint (integer2) or integer (integer4) for integer numbers (depending on the size of the number)

To initialize a column's value to null, specify the default value of null in any of the numeric conversion functions or the date function. Doing so makes the column nullable.

Important! Do not use null as a default value for character fields—this causes an attempt to create a character field of null length, which cannot be done, and returns an error.

Constraints

When you create or alter a table, define *constraints* for the table. Constraints are used to check for appropriate data values whenever data is entered or updated in the table.

Constraints are checked at the end of every statement that modifies the table. If the constraint is violated, the DBMS returns an error and aborts the statement. If the statement is in a multi-statement transaction, the transaction is not aborted.

Note: For other mechanisms used to ensure the integrity of data in a table, including integrities and rules, see the *Security Guide*.

In VDBA, define constraints using the Create Table or Alter Table dialog.

Constraint Types

There are several types of constraints:

- Unique
- Check
- Referential

Unique Constraints

You can define unique constraints at both the column and the table level. Columns that you specify as unique or that you use as part of a table-level unique constraint cannot be nullable. Column-level unique constraints ensure that no two rows in the table can have the same value for that column. At the table level, you can specify several columns, all of which are taken together to determine uniqueness.

For example, if you specify the department number and location columns to be unique at the table level, no two departments in the same location can have the same name. On the other hand, specifying the department name and location columns to be unique at the column level is more restrictive—in this case, no two departments can have the same name, regardless of the location, and no two locations can have the same name either.

There is a maximum of 32 columns that you can specify in a table-level unique constraint; however, a table can have any number of unique constraints.

In VDBA, define column-level unique constraints by enabling the Unique check box for the column in either the Create Table or the Alter Table dialog. You define table-level unique constraints using the Table Level Unique Constraint dialog.

Check Constraints

Check constraints ensure that the contents of a column fulfills user-specified criteria.

Specify check constraints by entering a Boolean expression involving one or more columns using the Table Level Check Constraint dialog in VDBA.

For example, enter the following expression to ensure that only positive values are accepted in the salary column:

```
salary > 0
```

The next example ensures that only positive values are accepted in the budget column and that expenses do not exceed the budget:

```
budget > 0 and expenses <= budget
```

Referential Constraints

Referential constraints are used to validate an entry against the contents of a column in another table (or another column in the same table), allowing you to maintain the referential integrity of your tables. You specify referential constraints by making selections in the Table References dialog in VDBA. For information on referential action options, see the *SQL Reference Guide*.

When defining a referential constraint, you must consider the following points:

- The table that you intend to reference must exist, with the appropriate primary key or unique constraint defined.
- Referencing columns from the table in which the constraints are being defined are compared to columns that make up the primary key or a table-level unique constraint in the referenced, or parent, table. The data types of the columns must, therefore, be comparable, and the referencing columns must correspond in number and position to those in the referenced table.
- You must have references permission for the referenced columns.

Example: Define a Referential Constraint

The following example of a referential constraint assumes that the employee table exists with a primary key constraint defined involving a name and an employee number column.

This example verifies the contents of the name and empno columns in the manager table against the primary key columns in the employee table, to ensure that anyone entered into the table of managers is on file as an employee.

To accomplish this, follow these steps:

1. Open the Table References dialog in VDBA.
2. Click New to create a new referential constraint, and optionally enter a new name for the constraint in the Constraint Name edit control.
3. Select the name column in the Table Columns list box, and click the double-right arrow (>>) to add the column to the Referencing Columns list box.
4. Select the empno column in the Table Columns list box, and click the double-right arrow (>>) to add the column to the Referencing Columns list box.
5. Select the employee table from the Reference to Parent Table drop-down list box.

   By default, the Primary Key option is selected and the primary key, which includes comparable name and employee number columns, appears in the edit control at the bottom of the box.
6. Click OK to add this referential constraint to the table definition.

Primary Key Constraint

An example of a referential constraint is a Primary key constraint.

Primary key constraints can be used to denote one or more columns, which other tables reference in referential constraints.

Note: Primary key constraints can be used as an alternative and slightly more restrictive form of unique constraint, but need not be used at all.

To define a primary key, you choose which columns are to be part of the key and assign to each a particular position in the key. Columns that are part of the primary key cannot be nullable, and the primary key is implicitly unique. A table can have only one primary key, which can consist of many columns.

In VDBA, you define primary key constraints using the Primary Key button in the Create Table or Alter Table dialog. Clicking this button opens the Primary Key dialog, where you can control which columns are part of the primary key, as well as the order of the columns in the primary key. For details, see online help.

Example: Define a Primary Key Constraint

For example, in the partnumbers table, define the partno column as the primary key. Assuming the inventory table had a comparable column named ipartno, define a referential constraint on the inventory table based on the partnumbers table.

To accomplish this, follow these steps:

1. Open the Table References dialog in VDBA.
2. Click New to create a new referential constraint, and optionally enter a new name for the constraint in the Constraint Name edit control.
3. Select the ipartno column in the Table Columns list box, and click the double-right arrow (>>) to add the column to the Referencing Columns list box.
4. Select the partnumbers table from the Reference to Parent Table drop-down list box.

   By default, the Primary Key option is selected and the partno column, which was previously defined as the primary key for the partnumbers table, appears in the edit control at the bottom of the box.
5. Click OK to add this referential constraint to the table definition.

In this case, the part numbers in the inventory table are checked against those in the partnumbers table. When defining this referential constraint, you did not have to specify the column to be referenced in the partnumbers table because it was defined as the primary key.

Indexes for Constraints

Special indexes are created whenever you specify a unique, primary key, or referential constraint for a table. No user—including the table owner—can explicitly drop these system-generated constraint indexes, because they are used internally to enforce the constraints.

For primary key and unique constraints, the index is built on the constrained columns as a mechanism to detect duplicates as rows are added to or updated in the table.

For referential constraints, the index is built on the referencing columns of the constraint. This index ensures that the internal procedures that enforce the constraint when a *referenced* row is deleted or referenced columns are updated can be executed efficiently. When a *referencing* row is inserted or referencing columns are updated, the unique constraint index built on the referenced columns is used to ensure the efficiency of enforcing the constraint.

Note: If you create an index, and then create a constraint that uses the index, the index cannot be dropped (but the constraint can be dropped). If you create a constraint using the WITH INDEX=*name* clause but do not create the index (which causes the system to generate the named index), and you drop the constraint, the index is also dropped, because the index is a system index and not a user index.

Options for Constraint Indexes

During table creation, specify options in VDBA for the constraint indexes, with similar options available when you alter a table. For example, the Table Level Unique Constraint dialog (accessible from both the Create Table and the Alter Table dialogs) has an Index button that allows you to fine tune the index used to enforce unique constraints. For additional information about the various dialog options, see online help.

These options give you more control over the index that is created, including the ability to specify:

- The location of the index

  Constraint indexes are, by default, stored in the default location for data files. Because of storage space or concurrency considerations, they can be stored in an alternate location.

- The storage structure type and other structure-specific characteristics

  By default, a B-tree storage structure is used for constraint indexes, but in some cases, a different structure can be more efficient. For more information on storage structures, see the chapter "Choosing Storage Structures and Secondary Indexes."

- That an existing secondary index be used instead of generating a new index

  You can save the overhead of generating an unnecessary index if you have an appropriate secondary index available. Simply indicate the name of the secondary index, and it is used to enforce the constraint instead of generating a new one.

  To use an existing secondary index for referential constraints, the referencing columns must match the first *n* columns of the index (although not necessarily in order).

  To use an existing secondary index for unique or primary key constraints, the index must be defined on exactly the same columns as the constraint, it must be a unique index, and it must specify that uniqueness is checked only after the UPDATE statement is completed.

  Note: Indexes enforcing uniqueness constraints in the ANSI/ISO style, as required by referenced columns of a referential constraint, must specify the "unique_scope = statement" option in the corresponding "create index" statement.

  For more information on creating a secondary index and specifying the scope for uniqueness checking, see the chapter "Choosing Storage Structures and Secondary Indexes" and online help.

- In the case of referential constraints, that no index be generated

  The index built for referential constraints is used only to improve the efficiency of the internal procedures that enforce the constraint when a referenced row is deleted or referenced columns are updated. Because the procedures can execute in its absence, the index is optional.

  In the absence of a referential constraint index, the internal procedures use a secondary index, if one is available that is defined on the referencing columns, and so the efficiency of the procedures can not be compromised. However, if no such secondary index exists, a full-table scan of the referencing table is necessary. Thus, choosing not to generate a referential constraint must be reserved for special circumstances, such as when:

  - An appropriate secondary index is available
  - Very few rows are in the referencing table (as in a prototype application)
  - Deletes and updates are rarely (if ever) performed on the referenced table

- That the base table structure be used for constraint enforcement

  This option requires a table that uses a keyed storage structure. Because heap, which is a non-keyed structure, is the default when you create a table, this option cannot be specified for constraints added at that time. The ability to use the base table structure for constraint enforcement is available only for constraints that are added when altering an existing table.

  Before you can specify the base table structure in lieu of a constraint index, you need to modify the table to change its storage structure type and to specify key columns to match the constraint columns it is used to enforce. If the base table structure is being used to enforce a primary key or unique constraint, you must also specify that uniqueness is checked only after the update statement is completed.

  Note: Indexes enforcing uniqueness constraints in the ANSI/ISO style, as required by referenced columns of a referential constraint, must specify the "unique_scope = statement" option in the corresponding "create index" statement.

  For more information on modifying a table’s storage structure, specifying key values, and specifying the scope for uniqueness checking, see the chapter “Maintaining Storage Structures.”

Delete Constraints

In VDBA, the Create Table dialog allows you to delete any constraint as you are designing your table, without restrictions. After you have saved the table, remove constraints using the Alter Table dialog; however, removing constraints is more complicated when altering a table, because of the possibility of dependent constraints.

For this reason, each dialog in VDBA that allows you to work with constraints gives you two delete options when altering a table. These same dialogs give you only one delete option when creating a table:

- Delete performs a restrictive delete, assuming that there are no dependent constraints. If you delete a constraint using this button, the constraint is dropped only if there are no dependent constraints; otherwise, if there are dependent constraints, the delete operation results in an error.
- Del Cascade performs a cascading delete by also dropping all dependent constraints. This is not available—and not needed—when creating a table.

For example, a unique constraint in one table can have a dependent referential constraint in another table. In this case, if you altered the table in which the unique constraint was defined and attempted to perform a Delete operation in the Table Level Unique Constraint dialog, it results in an error due to the existence of the dependent referential constraint. To delete the unique constraint successfully, use Del Cascade, which also deletes the referential constraint in the other table.

Note: In VDBA, column-level unique constraints are defined directly in the Create Table or Alter Table dialog. You cannot, however, remove a column-level unique constraint simply by disabling its Unique check box in the Alter Table dialog. To remove a column-level unique constraint, you must use the Table Level Unique Constraint dialog.

Techniques for Changing Table Columns

The examples here describe how to change table columns, including:

- Renaming a column
- Inserting a column
- Changing the data type of a column

There are no direct equivalents for changing columns in VDBA or in a single SQL statement.

Note: Renaming a column in a table or changing its data type does not change anything else that is dependent on the column. You need to update any objects, such as reports, forms, and programs, which are dependent on the old column name or data type. In addition, all of the procedures shown here require that you drop the original table, at which point certain other dependent objects, such as integrities, views, indexes, and grants, are also dropped. You must recreate these objects.

Important! We recommend that you back up your tables before changing them. If a problem occurs, you can then restore the original table and repeat the procedure. For additional information, see the chapter "Performing Backup and Recovery."

Example: Rename a Column

The following example renames two columns, name and addr, to employee and address. The salary column is not renamed. For assistance with any of these steps, see online help.

1. Assuming the table test already exists with columns name, addr, and salary, create a temporary table named temp.
2. Enable Create Table As Select in the Create Table dialog.
3. In the Select Statement edit control, enter the following select statement to rename the desired columns:

   ```
   select name as employee, addr as address, salary
       from test
   ```

4. Drop the original table, test.
5. Create a new table named test.
6. Enable Create Table As Select in the Create Table dialog.
7. In the Select Statement edit control, enter:

   ```
   select * from temp
   ```

8. Drop the temporary table, temp.

Be sure to update any objects, such as reports, forms, and programs that are dependent on the old column name, and recreate integrities, views, indexes, grants, and other dependent objects that were destroyed when the original table was dropped in Step 4.

Example: Insert a Column

When you add a column to an existing table using the Alter Table dialog, the column is placed after the last previously existing column in the table. To insert a new column between existing columns, you must follow a procedure similar to that outlined for renaming a column.

The following example illustrates inserting a new column, newcol, in the middle of an existing table with previously defined columns. If you need assistance with any of these steps, see online help.

1. Create a temporary table named temp.
2. Enable Create Table As Select in the Create Table dialog.
3. In the Select Statement edit control, enter the following select statement, inserting the new column:

   ```
   select col1, col2, varchar(' ') as newcol,
       col3, col4 from test
   ```

4. Drop the original table, test.
5. Create a new table named test.
6. Enable Create Table As Select in the Create Table dialog.
7. In the Select Statement edit control, enter:

   ```
   select * from temp
   ```

8. Drop the temporary table, temp.

Be sure to recreate integrities, views, indexes, grants, and other dependent objects that were destroyed when the original table was dropped in Step 4.

Note: To rearrange the current column order of a table without adding new columns, use this same procedure, selecting the columns in the desired order in Step 3.

Techniques for Moving a Table to a New Location

As a database grows, it can become necessary to move some of its tables to an alternate location. If a table has grown so large that you can no longer work with it at the current location, or the table needs to be distributed across multiple disks to improve performance, modify the table to use an alternate location.

Note: Before modifying the table, make sure you have met the requirements, as described in Requirements for Using an Alternate Location for a Table (see page 39).

You can modify a table's location using one of the following techniques:

- Relocate a table
- Reorganize a table

In VBDA, use the Modify Table Structure dialog. For a detailed procedure, see online help. For other uses of this dialog, see the chapter “Maintaining Storage Structures.”

Relocate a Table

Using the Modify Table Structure dialog in VDBA, you can move the data files for a table from one location to another. This is accomplished using the Locations button available when the Relocate radio button is enabled, which opens the Relocate Table dialog. For information on using this dialog, see online help.

Using this operation, it is possible to change one or more of the locations currently used by the table, without changing the number of locations used. For example:

- If a table is currently using a single location, choose a new location and the data files for the table are moved to the new location.
- If a table is currently using multiple locations, selectively specify which ones you want to change.

Reorganize a Table

You can increase or decrease the number of locations currently used by a table for its data files.

To do this, use the modify to reorganize SQL statement. In VDBA, use the Modify Table Structure dialog. Use the Locations button, which is available when the Change Locations radio button is enabled, which opens the Change Locations dialog. For specific information on using this dialog, see online help.

This operation requires more overhead than simply relocating a table because it performs a table reorganization in addition to moving files. Using this operation, you can:

- Expand a table that is currently using a single location to use multiple locations, including the option of no longer using the original location.
- Shrink a table that is currently using multiple locations to use a single location, including the option of no longer using any of its original locations.
- Reorganize a table that is currently using multiple locations to extend over a different number of locations, including the option of no longer using one or more of the original locations.

Afterwards, the table is reorganized to spread equally, in sized blocks, over the specified locations.

Assign an Expiration Date to a Table

By default, when you create a table, it has no expiration date. To give a table an expiration date, use the SAVE statement as described in the *SQL Reference Guide*.

A table is not automatically destroyed on its expiration date.

Purge Expired Tables

To purge expired tables, select Expired_Purge as the Operation in the Verify Database dialog in VDBA. For details on using this dialog, see online help.

Note: The Verify Database dialog has many other uses. For specific information on using this dialog, see online help.

Views

A *view* can be thought of as a virtual table. Only the definition for the view is stored, not the data. A table on which a view operates is called a *base table*.

A view definition can encompass 1 to 31 base tables. It can involve multiple tables joined together by their common columns using a where qualification.

A view can be created on other views or on physical database tables, including secondary indexes.

Primary uses for views include:

- Providing security by limiting access to specific columns in selected tables, without compromising database design
- Simplifying a commonly used query
- Defining reports

Because a view is a device designed primarily for selecting data, all selects on views are fully supported. Simply use a view name in place of a table name in the SELECT statement. Updating views is also supported (see Updates on Views), but updating a database by means of a view is not recommended.

Views and Permissions

Any user can create a view on any other user's tables or views, provided they have the permissions required to execute the SELECT statements that define the view. Any user can grant permissions on their views to any other user, provided they either own the base tables in the view or have the with grant option on the permissions they are granting. The granting of permissions is described in the *Security Guide*.

Working with View Objects

You can perform the following basic operations on views:

- Create view objects
- View existing view objects, including details such as the view definition, grantees, and rows
- Drop view objects

In SQL, you can accomplish these tasks using the SQL statements CREATE VIEW, HELP VIEW, and DROP VIEW. For details, see the *SQL Reference Guide*.

In VDBA, use the Views branch for a particular database in the Database Object Manager window.

Note: To drop a view the cache size that was needed to create the view must be enabled.

Updates on Views

Only a limited set of updates on views is supported because of problems that can occur. Updates are not supported on views that have more than one base table, or on any column whose source is not a simple column name (for example, set functions or computations). If the view was created using the With Check Option enabled in the Create View dialog, no updates or inserts are allowed on columns that are in the qualification of the view definition. For more information on the With Check Option control, see online help for the Create View dialog.

Updating is supported only if it can be guaranteed (without looking at the actual data) that the result of updating the view is identical to that of updating the corresponding base table.

Note: Updating, deleting, or inserting data in a table using views is not recommended. You cannot update or insert into a view with Query-By-Forms. You can update, delete, or insert with SQL statements, but you must abide by the following rules, keeping in mind that an error occurs if you attempt an operation that is not permitted.

Types of Updates Not Permitted on Views

You cannot perform the following types of updates on a view:

- One that involves a column that is a set function (aggregate) or derived from a computational expression

  In the following example of a SELECT statement used to define a view, you cannot update the tsal column because it is a set function:

  ```
  select dept, sum(sal) as tsal
       from deptinf group by dept
  ```

- One that causes more than one table to be updated

  Consider the following example of a SELECT statement used to define a view:

  ```
  select e.name, e.dept, e.div, d.bldg
    from emp e, deptinf d
    where e.dept = d.dname and e.div = d.div
  ```

  Updates to this data must be done through the underlying base tables, not this view.

- You can update a column that appears in the qualification of a view definition, as long as the update does not cause the row to disappear from the view. For example, if the WHERE clause is as follows, update the deptno from 5 to 8, but not from 5 to 20:

  ```
  where deptno < 10
  ```

Schemas

A *schema* is a collection of any of the following database objects:

- Tables
- Views
- Grants

Each user can have only one schema consisting of definitions of the above types of database objects that the user owns. The database objects belong to the specific schema.

By default, the current user’s schema is always assumed.

Tools for Managing Schemas

You can manage schemas directly using the SQL statement, CREATE SCHEMA. This statement allows you to create a schema, create the tables and views in that schema, and grant appropriate permissions as a unit. For more information on this statement, see the *SQL Reference Guide*.

Various SQL statements allow you to specify a schema name to indicate a table or view belonging to a schema other than the one associated with the current user, as described in the *SQL Reference Guide*.

In VDBA, a schema is created for you automatically, and objects that you create are added to your schema. View the contents of your schema using the Schemas branch for a particular database in the Database Object Manager window.

Synonyms, Temporary Tables, and Comments

The following features are available to the DBA and other users to assist in manipulating table data and referencing tables:

- Synonyms for table names
- Temporary tables local to an individual session
- Comments for documenting and describing tables

Synonyms

The DBA or any user is allowed to create synonyms for tables, views, and indexes. These alternate names, or aliases, can be used to define shorthand names in place of long, fully qualified names.

After a synonym is created, it can be referenced the same way as the object for which it was created. For example, if you create a synonym for a table or view, issue SELECT statements using the synonym name, just as you use the table or view name.

Working with Synonym Objects

You can perform the following basic operations on synonyms:

- Create synonym objects
- View existing synonym objects, including the detailed properties of each individual object
- Drop synonym objects

In SQL, you can accomplish these tasks using the statements CREATE SYNONYM and DROP SYNONYM. For details, see the *SQL Reference Guide*.

In VDBA, use the Synonyms branch for a particular database in the Database Object Manager window.

Temporary Tables

Temporary tables are useful in applications that need to manipulate intermediate results and minimize the processing overhead associated with creating tables.

Temporary tables reduce overhead in the following ways:

- No logging or locking is performed on temporary tables.
- No page locking is performed on temporary tables.
- Disk space requirements are minimized. If possible, the temporary table is created in memory and never written to disk.
- No system catalog entries are made for temporary tables.

Because no logging is performed, temporary tables can be created, deleted, and modified during an online checkpoint.

Temporary tables are:

- Visible only to the session that creates them
- Deleted automatically when the session ends
- Declarable by any user, whether or not the user has the create_table permission

The DECLARE GLOBAL TEMPORARY TABLE statement is used to create temporary (session-scope) tables. In VDBA, use the Create Table dialog.

All temporary tables are automatically deleted at the end of the session. To delete a temporary table before the session ends, issue a DROP TABLE statement.

Temporary Table Declaration and the Optional SESSION Schema Qualifier

The DBMS Server supports two syntaxes for declaring and referencing global temporary tables:

With the SESSION Schema Qualifier

If the DECLARE GLOBAL TEMPORARY TABLE statement defines the table with the SESSION schema qualifier, then subsequent SQL statements that reference the table must use the SESSION qualifier.

When using this syntax, the creation of permanent and temporary tables with the same name is allowed.

Without the SESSION Schema Qualifier

If the DECLARE GLOBAL TEMPORARY TABLE statement defines the table without the SESSION schema qualifier, then subsequent SQL statements that reference the table can optionally omit the SESSION qualifier. This feature is useful when writing portable SQL.

When using this syntax, the creation of permanent and temporary tables with the same name is not allowed.

Note: In both modes, a session table is local to the session, which means that two sessions can declare a global temporary table of the same name and they do not conflict with each other.

Note: Syntaxes cannot be mixed in a single session. For example, if the table is declared with SESSION the first time, all declarations must use SESSION.

Examples of Working with Temporary Tables

To create two temporary tables, names and employees, for the current session, issue the following statements:

```
declare global temporary table session.names
  (name varchar(20), empno varchar(5))
  on commit preserve rows
  with norecovery
declare global temporary table session.employees as
  select name, empno from employees
  on commit preserve rows
  with norecovery
```

Note: The "session." qualifier in the example is optional. If omitted, the name of the temporary table cannot be the same as any permanent table names.

The names of temporary tables must be unique only in a session.

For more information on working with temporary tables, see the descriptions for DECLARE GLOBAL TEMPORARY TABLE and DROP statements in the *SQL Reference Guide*.

Comments to Describe Tables and Views

When using VDBA, tables and views are self-documenting. For example, you can see the definition of a view at a glance, as well as its rows and the grants that have been defined. For a table, you can view its rows and columns, as well as properties, statistics, and other pertinent information.

When working with tables and views in applications, however, it is helpful to include commentary about the structure and uses of tables and views.

Tables and views can be commented with:

- Comment lines in SQL or a host language, for example, "/*comment*/" or "--comment"
- Comments specified with the COMMENT ON statement
- Comments in embedded SQL (ESQL) programs specified with the DECLARE TABLE statement

The Comment On Statement

The COMMENT ON statement allows you to add commentary to SQL programs. Using this statement, you can create a comment for the table as a whole and for individual columns in the table.

For example, to add a remark on the name column and on the status column of the employee table:

```
comment on column employee.name is
  'name in the format: lastname, firstname';

comment on column employee.status is
  'valid codes are:
     01, exempt; 02, non-exempt; 03, temp';
```

To delete a comment, specify an empty string. For example, the comment on the status column can be deleted by the statement:

```
comment on column employee.status is '';
```

For complete details, see the comment on statement in the *SQL Reference Guide*.

The Declare Table Statement

The DECLARE TABLE statement can be used to describe the structure of a table in ESQL programs. With this statement, you can document the columns and data types associated with a table.

For example, the employee table can be described with a DECLARE TABLE statement as follows:

```
exec sql declare userjoe.employee table
         (
         emp_number    integer2 not null not default,
         last_name     varchar(20) not null,
         first_name    varchar(20),
         birth_date    date not null not default
         );
```

For complete details, see the entry for the DECLARE TABLE statement in the *SQL Reference Guide*. For information on ESQL programs, see the chapter “Embedded SQL” in the *SQL Reference Guide*.

Chapter 4: Populating Tables

This section contains the following topics:

This chapter describes how to use the COPY statement, which is the fastest and most flexible method of loading data into tables. It also discusses the fastload operation and the SET NOLOGGING statement as other methods of loading data.

Methods of Populating Tables

You can load data into tables using the following methods:

- COPY statement

  The COPY statement is a good method to use for loading large quantities of data quickly from files, and is flexible in dealing with record formats.

- INSERT statement

  Use the INSERT statement to enter a small amount of data.

- Append mode of Query-By-Forms (QBF)

  Use QBF for interactive data entry.

The INSERT statement and the QBF append mode, as alternatives to the COPY statement, provide the most potential for customized validity checking and appending to more than one table at a time. For more information on these alternatives, see the *SQL Reference Guide* and the *Character-based Querying and Reporting Tools User Guide*, respectively.

Copy Statement Syntax

To load data from a table into a file or from a file into a table, use the COPY statement.

Each COPY statement must specify only one table name. An optional schema name can be specified. The TABLE keyword is optional and can be included for readability.

The table name is followed by a list in parentheses containing none, one, or more column format specifications, up to the total number of columns in the table. The column specifications depend on the type of copy being performed.

For details on the COPY statement, see the *SQL Reference Guide*.

Copy Into (Unload Data) and Copy From (Reload Data)

The COPY statement is bidirectional; it unloads data from a table and loads data into a table.

The INTO and FROM keywords specify the direction of data movement:

- The COPY INTO statement performs the unload operation, copying the data **into** a file from the database. For example, to copy the horx table into the pers.data file, use the following statement:

  ```
  copy table horx (name=char(20), code=integer)
      into 'pers.data';
  ```

- The COPY FROM statement performs the reload operation, copying the data **from** a file into the database. For example, to copy the agent table from the myfile.in file into the database, use the following statement:

  ```
  copy table agent (ano=integer, code=integer2)
      from 'myfile.in';
  ```

A common use of COPY is to unload a table for backup to tape or for transfer to another database. In either case, there is the possibility of reloading the data into a database later.

File Name Specification on the Copy Statement

Only one file can be specified in the copy operation. If the file does not exist when copying to a file, it is created.

Windows: If the file exists, copy overwrites it. When specifying a file name, always enclose it in single quotation marks. Omit the drive and directory portion of the file name if the file is in the current directory. Otherwise, provide a full path name with drive and directory. The following example shows a full path name; this is an example of Binary Copying (see page 75):

```
copy emp() from 'D:\users\fred\emp.lis’;
```

UNIX: If the file exists, copy overwrites it. When specifying a file name, always enclose it in single quotation marks. Omit the full path name if the file is in the current directory. Otherwise, provide a full path name. The following example shows a full path name; this is an example of Binary Copying (see page 75):

```
copy emp() from '/usr/fred/emp.lis';
```

Important! *The COPY statement is not able to expand $HOME or recognize the UNIX variables set in your environment. Do not use these variables to specify a path name for the COPY statement. For example, the following COPY statements do not work*:

```
copy emp () from '~fred/emp.lis';     /* invalid */
copy emp () from '$HOME/emp.lis';     /* invalid */
```

VMS: If the file exists, COPY creates another version of the file. When specifying a file name, you can optionally give a VMS file type:

```
into | from 'filename[, type]'
```

where *type* is text, binary, or variable.

With-Clause Options of the Copy Statement

A *with-clause* can be used to further describe and control the copy being performed. For a description of the syntax for the with-clause, see the *SQL Reference Guide*. For valid with-clause options, see the *Command Reference Guide*.

Copy Statement Operation

The COPY statement allows you to do the following:

- Copy all of a table or selected columns
- Rearrange columns
- Manipulate data formats of the column data

Binary and Formatted Copying

A special form of the COPY statement is used to perform a fast copy of an entire table with no format changes.

- A binary copy is a fast method of copying an entire table.

  No column names are specified in a binary copy. For example:

  ```
  table emp ()
  ```

  The entire table is moved, byte for byte, with no record delimiters or data type conversions.

- A formatted copy is performed on selected columns of data (which can include all columns), with type conversions being performed as necessary during the copy.

  One or more column names are specified in a formatted copy. For example:

  ```
  table emp (eno=integer, ename=char(10))
  ```

  Although this type of copy is not as fast as a binary copy, it allows you to fully control the columns copied, the column order, and the data type conversions to be performed.

Bulk and Incremental Copy

When copying data to a table, the COPY FROM statement can run in either of the following modes:

- Bulk copy

  A bulk copy is a copy operation optimized for speed that allows the DBMS Server to exploit group writes and minimal transaction logging. Bulk copying is done whenever the characteristics of the target table allows. The bulk copy can be performed from any source file, either binary or formatted.

  Doing a bulk copy from a binary file is the fastest method to copy large amounts of data.

- Incremental copy

  In incremental mode, data is added to the table using inserts, causing single-page writes and full transaction logging. Incremental mode is used for the copy whenever the characteristics of the target table do not allow a bulk copy to be performed.

Copy Permissions and Integrities

The COPY statement itself does not require permissions to run. However, you must have permission to access the table being copied. At least one of the following must apply:

- You own the table.
- You have been granted select (for COPY INTO) or insert (for COPY FROM) privilege on the table.
- The table has select (for COPY INTO) or insert (for COPY FROM) permission granted to public.

To copy data in and out of the database as quickly as possible, the COPY statement does not check for:

- Permissions other than select/insert as described above
- Integrities

  When copying data into a table, copy ignores any integrity constraints (defined with the CREATE INTEGRITY statement) against the table.
- Constraints

  When copying data into a table, COPY ignores ISO Entry SQL92 check and referential constraints (defined using the CREATE TABLE and ALTER TABLE statements), but does not ignore unique and primary key constraints.
- Rules

  The COPY statement does not fire any rules defined against the table.

For information on integrities, constraints, and rules, see the chapter "Ensuring Data Integrity."

Locking During a Copy

When you use the COPY statement, the locking system takes one of the following actions:

- A Shared lock on the table while data is being copied into a file
- An Exclusive lock on the table during bulk copies to the table, for maximum speed
- An Intended Exclusive lock on the table during an incremental copy to the table. Because inserts are used to update the table, the copy can encounter lock contention.

For a complete explanation of locking, see the chapter "Understanding the Locking System."

Binary Copying

The binary copy is designed for reloading data into tables that have exactly the same record layout as those from which the data was unloaded. Those tables must be on a machine with the same architecture as that from which they were unloaded.

The utilities designed for quick unloading and reloading of tables or databases—copydb and unloaddb (and their VDBA equivalent features)—both use the binary form of COPY by default. They automate the process so you can easily recreate and reload tables of identical layout. For information on copydb and unloaddb, see the *Command Reference Guide*.

Copy Data into a Binary File

The quickest form of the COPY statement to code and execute for unloading data into a file is a binary copy. A binary copy always creates a file of type binary. This statement has the following format:

```
COPY [TABLE] [schema.]tablename () INTO 'output_filename'
  [standard-with-clauses]
```

You omit the column list from the COPY statement. For example, the following statement copies the entire dept table into the dept_file file in binary format:

```
copy dept () into 'dept_file';
```

The binary COPY INTO statement copies all rows of a table to the output file in a proprietary binary format, using the order and format of the columns in the table. It places no delimiters between fields or between records. It performs no type conversions; all data items retain the type they had in the table.

Unloading data in binary format for backup can be inconvenient if you need to inspect the file data later.

Note: If any columns have a type other than character, they are not readable as characters in the output file.

If you unload a table in binary format, the data must subsequently be reloaded in binary format. You cannot use unloaded binary data to later load tables that have a different column order, number of columns, data types, or table structure. To perform these functions, use formatted COPY statements.

Reload a Table in Binary Format

Use the following form of the COPY statement to reload a table from a file containing data in binary format:

```
COPY [TABLE] [schema.]tablename () FROM 'output_filename'
  [standard-with-clauses]
  [bulk-copy-with-clauses]
```

This form of the COPY FROM statement must be used to reload from a binary file (that is, one created with an empty column list in the COPY INTO statement).

For example, the following statement copies the binary data from the dept_file file into the new_dept_table table:

```
copy new_dept_table () from 'dept_file'
```

The table is recreated with the same column and data type format specifications as in the original table. If the table characteristics allow, include bulk copy (see page 84) WITH clauses.

Formatted Copying

Formatted copying provides a flexible means of copying tables.

Column Name and Format Specifications

When using the COPY statement to do formatted copying, specify the column name and the format in which that column's data is to be copied, as follows:

```
column_name = format [null-clause]
```

where:

column_name

Specifies the column from which data is read or to which data is written. The name in the copy target must be the same as in the copy source; you cannot change column names in the COPY statement.

format

Specifies the storage format in which the column values are stored and delimited. The storage format is based on the data type. The COPY statement can copy all data types except logical keys.

The column names and their formats must be separated by commas, and the list must be in parentheses.

Summary of Data Types and Storage Formats

A summary of data types and their storage format characteristics is given in the table below. For detailed information on storage formats and data conversions of the various data types, see the *SQL Reference Guide*.

Class	**Data Types**	**Description and Copy Notes**
Character data	char	Fixed-length strings with blank padding at the end. ▪ char(0)[*delim*] copies the string without requiring a specified length, with a default or specified delimiter as an end of record. ▪ char(*n*) copies *n* = 1 to x characters. "x" represents the lesser of the maximum configured row size and 32,000.
Character data	varchar	Variable-length strings preceded by a length. ▪ varchar(0) copies the string and its stored length. ▪ varchar(*n*) copies *n* = 1 to x characters and its stored length, with null padding at the end. "x" represents the lesser of the maximum configured row size and 32,000.
Character data	long varchar	Stored in segments and terminated by a zero length segment. Each segment is composed of an integer specifying the length of the segment, followed by a space and the specified number of characters. The end of the column data is specified through a termination, zero length segment (that is, an integer 0 followed by a space). The following example shows two data segments, followed by the termination zero length segment. The first segment is 5 characters long, the second segment is 10 characters long, and the termination segment is 0 character long. The maximum length of each segment is 32737. 5 abcde10 abcdefghij 0 (with a space after the terminating 0 character) (In this example, the effective data that was in the column is abcdeabcdefghij) If the long varchar column is nullable, specify the WITH NULL clause. An empty column is stored as an integer 0, followed by a space.

Class	Data Types	Description and Copy Notes
Unicode data	nchar	Fixed-length Unicode strings in UTF-8 format (padded with blanks if necessary).
Unicode data	nvarchar	Variable-length Unicode string in UTF-8 format preceded by a 5-character, right-justified length specifier.
Unicode data	long nvarchar	Stored in segments, and terminated by a zero length segment. Each segment is composed of an integer specifying the length of the segment, followed by a space and the specified number of Unicode characters in UTF-8 format. The end of the column data is specified through a termination, zero length segment (that is, an integer 0 followed by a space). The maximum length of each segment is 32727 bytes. **Note:** The "number" of Unicode characters in a segment will be less than 32767. For example, each UTF-16 character in Basic multilingual plane can occupy 1 to 3 bytes in UTF-8 format. If the long nvarchar column is nullable, specify the WITH NULL clause. An empty column is stored as an integer 0, followed by a space. The UTF-8 encoded long nvarchar data segments are similar to long varchar data segments. For an example of the encoded data segment, see the description for long varchar.
Binary data	byte	Fixed-length binary data with padding of zeroes at the end. ▪ byte(0)[*delim*] copies the data without requiring a specified length, with a default or specified delimiter as an end of record. ▪ byte(*n*) copies *n*= 1 to x bytes. “x” represents the lesser of the maximum configured row size and 32,000.
Binary data	byte varying	Variable-length binary data preceded by a length. ▪ byte varying(0) copies the data and its stored length. ▪ byte(*n*) copies *n*= 1 to x bytes and its stored length, with zero padding. “x” represents the lesser of the maximum configured row size and 32,000.

Class	Data Types	Description and Copy Notes
Binary data	long byte	Binary data stored in segments, and terminated by a zero length segment. Each segment is composed of an integer specifying the length of the segment, followed by a space and the specified number of characters. The end of the column data is specified through a termination, zero length segment (that is, an integer 0 followed by a space). The following example shows two data segments, followed by the termination zero length segment. The first segment is 5 characters long, the second segment is 10 characters long, and the termination segment is 0 character long. The maximum length of each segment is 32737. 5 abcde10 abcdefghij 0 (with a space after the terminating 0 character) (In this example, the effective data that was in the column is abcdeabcdefghij) If the long byte column is nullable, specify the WITH NULL clause. An empty column is stored as an integer 0 followed, by a space.
Numeric data	integer1	Integer of 1-byte length (-128 to +127).
Numeric data	smallint	Integer of 2-byte length (-32,768 to +32,767).
Numeric data	integer	Integer of 4-byte length (-2,147,483,648 to +2,147,483,647).
Numeric data	decimal	Fixed-point exact numeric data, up to 31 digits. Range depends on precision and scale. Default is (5,0): -99999 to +99999.
Numeric data	float4	Single precision floating point number of 4-byte length (7 digit precision), -1.0e+38 to +1.0e+38.
Numeric data	float	Double precision floating point number of 8-byte length (16 digit precision), -1.0e+38 to +1.0e+38).
Date/time data	ansidate	4-byte binary
Date/time data	time	8- or 10-byte binary
Date/time data	timestamp	12- or 14-byte binary
Date/time data	interval year to month	3-byte binary

Class	Data Types	Description and Copy Notes
Date/time data	interval day to second	12-byte binary
Abstract data types	ingresdate	Date of 12-byte length, 1-jan-1582 to 31-dec-2382 (for absolute dates) and -800 years to 800 years (for time intervals).
Abstract data types	money	Exact monetary data of 8-byte length, $-999,999,999,999.99 to $999,999,999,999.99
Copy statement only	d	Dummy field. ▪ d0*delim* on COPY INTO copies the delimiter into the (empty) field. ▪ d0[*delim*] on COPY FROM skips the data in the field, up to the default or specified delimiter. ▪ d*n* on COPY INTO copies the name of the column *n* times. On COPY FROM, skips the field of *n* characters.
User-defined data types (UDTs)		Use char or varchar.

Copy Statement and Nulls

When you copy data from a table to a file or vice versa, the WITH NULL clause of the COPY statement allows you to substitute a value for nulls.

When you use variable length data formats when copying, you **must** replace the null values with some string that represents nulls; for example:

```
copy table personnel (name=char(20),
    salary=char(0) with null ('N/A'),
    dummy=d0nl)
    into 'pers.data';
```

After executing this statement, the pers.data file contains “N/A” for each null salary.

With other data formats, you are not required to substitute a value for nulls. However, if you do not, your file contains unprintable characters.

When substituting a value for nulls, the value:

- Must not be one that occurs in your data
- Must be compatible with the format of the field in the file:
  - Character formats require quoted values
  - Numeric formats require unquoted numeric values

Do not use the word NULL if you are copying to a numeric format. The file does not accept an actual null character or the word NULL for numeric format.

Copy Data into a Formatted File

Use the following COPY statement to copy table data into a formatted file:

```
COPY [TABLE] [schema.]tablename
  ([column_name = format [WITH NULL [(value)]]
  {, column_name = format [WITH NULL[(value)]]}])
  INTO 'output_filename' [standard-with-clauses]
```

One or more column names appear, with format specifications. The column names must be the same as those in the table. However, the order of the columns can be different from the order in which they are currently stored in the table (except as noted below). Also, the format does not have to be the same data type or length as their corresponding entries in the table. The data is copied with any column reorganization or format conversions being made as necessary.

Note: When copying from a table that includes long varchar or long byte columns, you must specify the columns in the order they appear in the table.

Two major categories of data that can be unloaded into files are fixed-length fields and variable-length fields.

Data with Fixed-Length Fields

Fixed-length fields can use implicit or explicit specification of the field length.

- If you use the (0) notation for fixed-length character or byte data, the length is implicitly specified. For example, if you use char(0), character columns are copied into the file using the full length of the column.
- Columns containing numeric data (such as integer or float data types) can be explicitly formatted using the -i or -f SQL option flags. For details on these flags, see the sql command description in the *Command Reference Guide.*
- If you use a length specifier, the field length is explicit. For example, for char(*n*), the COPY INTO statement stores exactly n characters. Excess characters are discarded and shorter columns are padded with blanks.
- The varchar(*n*) format stores exactly *n* characters with a leading length indicator in ASCII format, discarding any excess characters. Shorter columns are padded with null bytes.

Data with Variable-Length Fields

Variable-length data items are written to a file by the COPY statement with the formats:

varchar(0)
long varchar(0)
byte varying(0)
long byte(0)
nvarchar(0)
long nvarchar(0)

An ASCII length is written preceding the data. The length of the data copied corresponds to the number of characters or bytes in the column, not the width of the column specified in the CREATE statement. Varchar(0) compresses the data, whereas char(0) does not.

Reload Formatted Data

Use the following form of the COPY statement to reload a table from a file containing formatted data:

```
COPY [TABLE] [schema.]tablename
  ([column_name = format [WITH NULL [(value)]]
  {, column_name = format [WITH NULL[(value)]]}])
  FROM 'input_filename' [standard-with-clauses]
  [bulk-copy-with-clauses]
```

The input file name can contain user-created data for reading in new data to a table, or a formatted file created by a COPY INTO statement. You must specify the column names in sequence according to the order of the fields in the formatted file (that is, the same order in which they appeared in a COPY INTO statement). The format does not have to be the same data type or length as their corresponding entries in the file. The target table can be empty or populated; in the latter case, the COPY FROM operation merges the new data from the file with the existing table data. If the table characteristics allow, include bulk copy (see page 84) WITH clauses.

Bulk Copy

The COPY statement to reload a table whose characteristics allow a bulk copy to be performed has the following format:

```
COPY [TABLE] [schema.]tablename
  ([column_name = format [WITH NULL [(value)]]
  {, column_name = format [WITH NULL null[(value)]]}])
  FROM 'input_filename'
  [WITH [standard-with-clauses]
  [, allocation = n] [, EXTEND = n] [, ROW_ESTIMATE = n]
  [, FILLFACTOR=n] [, MINPAGES=n] [, MAXPAGES=n]
  [, LEAFFILL=n] [, NONLEAFFILE=n]]
```

If the file is formatted, you must specify columns in the column list as described for reloading formatted data. If the file is binary, use an empty column list.

Bulk Copying Requirements

To perform a bulk copy when loading data from a file into a table, the table must have the following characteristics:

- The table has no secondary indexes.
- The table is not journaled.
- The table is not partitioned.
- The table is either a heap table (the data from the file is appended to the end of the heap table) or empty and less than 18 pages in size if the table is hash, B-tree, or ISAM (the table is rebuilt with the new data from the file).

If these requirements are not met, the copy is performed in incremental mode.

Transaction Logging During Bulk and Incremental Copy

The bulk copy requires only minimal transaction logging.

Note: A transaction is still entered into the log file and normal logging occurs for the associated system catalogs.

In contrast, an incremental copy can generate a large amount of transaction log records. The incremental copy requires logging for every record transfer and every structural change to the table.

An alternative method to reducing logging is described in Large Data Loads with the Set Nologging Statement (see page 98).

Bulk and Incremental Copy Processing

The processing for a bulk copy is similar to a MODIFY statement, except that the data comes from an external source rather than an existing table. For a bulk copy, the following occurs:

1. The COPY statement reads all data from the source.
2. The COPY statement deposits the data in the Data Manipulation Facility (DMF) sorter.
3. The sorter sorts all data into the required order.

   Note: For a heap table, no sorting is done.
4. The COPY extracts the data from the sorter, builds the table, and populates the table.

In contrast, for an incremental copy, the sequence is as follows:

1. Reads one record from the external file.
2. Adds to the table as an insert.
3. Repeats these steps until the data has been copied.

Bulk Copy With-Clauses

The WITH clause options on the COPY statement for bulk copy operate like the corresponding clauses in the MODIFY statement.

If these clauses are omitted, the table default values in the system catalogs are used. If any of these clauses are specified, the values become the new defaults for the table in the system catalogs.

The following clauses can be used only with a bulk copy:

Note: If these clauses are used with a COPY statement with columns specified, an error message is returned and the copy is not performed.

- WITH ALLOCATION

  A bulk **copy from** can preallocate table space with the allocation clause. This clause specifies how many pages are preallocated to the table. This clause can be used only on tables with B-tree, hash, or ISAM storage structures.For example, preallocate 1000 pages for bulk copying from the emp.data file into the emp table:

  ```
  copy emp() from 'emp.data' with allocation = 1000;
  ```

  VMS: Preallocating space with the allocation clause is important particularly in VMS installations to increase loading efficiency. ▼

- WITH EXTEND

  A bulk copy from can extend table space with the extend clause. This clause specifies how many pages the table is extended. This clause can be used only on tables with B-tree, hash, or ISAM storage structures.

  For example, extend table emp by 100 pages for bulk copying from the emp.data file:

  ```
  copy emp() from 'emp.data' with extend = 100;
  ```

- WITH ROW_ESTIMATE

  A bulk copy can specify an estimated number of rows to be copied during the bulk copy. It can be a value from 0 to 2,147,483,647 ((231-1). This clause can be used only on tables with B-tree, hash, or ISAM storage structures.

  For example, set the row estimate on the emp table to one million for bulk copy from the emp.data file:

  ```
  copy emp() from 'emp.data' with row_estimate = 1000000;
  ```

Providing a row estimate can enhance the performance of the bulk copy by allowing the sorter to allocate a realistic amount of resources (such as in-memory buffers), disk block size, and whether to use multiple locations for the sort. In addition, it is used for loading hash tables in determining the number of hash buckets. If you omit this parameter, the default value is 0, in which case the sorter makes its own estimates for disk and memory requirements.

To obtain a reasonable row estimate value, use known data volumes, the HELP TABLE statement, and information from the system catalogs. For more information, see the chapter "Maintaining Storage Structures." An over-estimate causes excess resources of memory and disk space to be reserved for the copy. An under-estimate (the more typical case, particularly for the default value of 0 rows) causes more sort I/O to be required.

- WITH FILLFACTOR

  A bulk copy from can specify an alternate fillfactor. This clause specifies the percentage (from 1 to 100) of each primary data page that must be filled with rows during the copy. This clause can be used only on tables with B-tree, hash, or ISAM storage structures.

  For example, set the fillfactor on the emp table to 10% for bulk copy from the emp.data file:

  ```
  copy emp() from 'emp.data' with fillfactor = 10;
  ```

- WITH LEAFFILL

  A bulk copy from can specify a leaffill value. This clause specifies the percentage (from 1 to 100) of each B-tree leaf page that must be filled with rows during the copy. This clause can be used only on tables with a B-tree storage structure.

  For example, set the leaffill percentage on the emp table to 10% for bulk copy from the emp.data file:

  ```
  copy emp() from 'emp.data' with leaffill = 10;
  ```

- WITH NONLEAFFILL

  A bulk copy from can specify a nonleaffill value. This clause specifies the percentage (from 1 to 100) of each B-tree non-leaf index page that must be filled with rows during the copy. This clause can be used only on tables with a B-tree storage structure.

  For example, set the nonleaffill percentage on the emp table to 10% for bulk copy from the emp.data file:

  ```
  copy emp() from 'emp.data' with nonleaffill = 10;
  ```

- WITH MINPAGES, MAXPAGES

  A bulk copy from can specify minpages and maxpages values. The MINPAGES clause specifies the minimum number of primary pages that a hash table must have. The MAXPAGES clause specifies the maximum number of primary pages that a hash table must have. This clause can be used only on tables with a hash storage structure.

  If these clauses are not specified, the primary page count for the bulk copy is determined as follows:

  – If the COPY statement has a ROW_ESTIMATE clause, that size, along with the row width and fill factor, is used to generate the number of primary pages.

  – Otherwise, the table's default in the system catalogs is used.

  The following example sets the number of primary data pages (hash buckets) for bulk copying from the emp.data file into the emp table:

  ```
  copy emp() from 'emp.data' with minpages = 16384, maxpages = 16384
  ```

For further details on these WITH clause options, see the chapter "Maintaining Storage Structures."

Example: Perform a Bulk Copy to Create a Hash Table

The following sequence of statements allows a bulk copy to be performed. This example creates a hash table:

```
create table tmp1
    (col1 integer not null,
    col2 char(25))
    with nojournaling;
    modify tmp1 to hash;
    copy tmp1() from 'tmp1.saved'
    with row_estimate = 10000,
    maxpages = 1000,
    allocation = 1000;
```

Bulk copy is chosen for the copy because the table is not journaled (it is created with nojournaling), it has no indexes (none have yet been created), and the table has under 18 pages (it is a newly created table that has not yet been populated with data).

The MODIFY...TO HASH operation is quick because the table is empty at this point. The ROW_ESTIMATE parameter allows a more efficient sort than if the default estimate (of 0 rows) is used. Additionally, for the hash table, ROW_ESTIMATE enables the number of hash buckets to be calculated efficiently. This calculation uses the row width (set by the CREATE TABLE statement), ROW_ESTIMATE , MAXPAGES, and the (default) FILLFACTOR. The COPY statement includes an ALLOCATION clause to preallocate disk space for the table to grow, increasing efficiency of later row additions.

Example: Perform Bulk Copy and Create B-tree Table

The following example of a bulk copy uses a B-tree table:

```
create table tmp2
    (col1 integer not null,
    col2 char(25));. . .
    Populate table. . .
    Save any data needed in table
    modify tmp2 to truncated;
    modify tmp2 to btree;
    set nojournaling on tmp2;
    copy tmp2() from 'tmp2.saved'
    with row_estimate = 10000,
    leaffill = 70,
    fillfactor = 95,
    allocation = 1000;
```

The existing table tmp2 is truncated to assure it has fewer than 18 pages. This also removes any indexes. Journaling is disabled for the table with the SET NOJOURNALING statement. The table meets the bulk copy requirements and a bulk copy is performed.

The COPY statement includes a row estimate for efficient sorting during the copy. The LEAFFILL and FILLFACTOR clauses allow the B-tree table to be set up as specified during the copy operation. The allocation clause provides for efficient future table growth.

Example: Perform Bulk Copy into a Heap Table

When a heap table is unjournaled and has no indexes, all copies are performed using bulk copy, regardless of the size of the table. Bulk copying into a non-empty heap table is allowed by logging the last valid page before starting the copy. If an error or rollback occurs, all new pages are marked as free.

Note: The table is not returned to its original size.

For example, in the following sequence of statements, all of the copy operations into the heap table are done as bulk copies:

- Create a heap table
- Copy to the heap table (table is empty)
- Copy to the heap table (append to the table)
- Perform inserts, updates, and deletes
- Copy to the heap table (append to the table)

Fastload Operation

The fastload operation loads data from a binary format file into a single table in a single database. It loads each contiguous *n* bytes of data into a new row in the target table.

The fastload operation can be performed using the fastload command (see the *Command Reference Guide*) or in VDBA using the Fastload Table dialog. In VDBA, you select the desired table from a database and choose a file from which you want to load the data in the Fastload Table dialog.

Requirements for Using Fastload

The following requirements must be met to perform a fastload:

- Since the fastload command creates its own server, if simultaneous access to the database is required for additional sessions, the following DBMS parameter changes are required for any DBMS that has access to the same database:
  - cache_sharing ON
  - fast_commit OFF
  - sole_server OFF
- The fastload operation must be able to obtain an exclusive lock on the table or fastload exits.
- The file's data format must match the table's data format.

  If the formats do not match, incorrect data are loaded into the table. For example, if each record in a file contains a 5-byte char and a 4-byte integer—and this file is loaded into a table that has a 4-byte char field followed by a 4-byte integer field—fastload reads 8-bytes of the file and load it as a row into the table. This means that the integer field does not contain the actual integer in the original file because the last byte of the 5-byte char field plus 3-bytes of the integer field are interpreted as a 4-byte integer. The problem grows as more data is read because the data are off by one more byte for each row.

  Check that the record length in the file matches the record length expected by the table. For tables that do not include large object columns (such as long varchar and long byte), the record length should match the row width, as given by the SQL HELP TABLE statement.

  In many cases, fastload is unable to determine the record size of a binary file (this is the case on all UNIX platforms); in these cases, fastload generates a message warning that no format checking can be performed. The warning also contains the expected size of records in the binary file.

- Be aware of the extra data added by the Ingres varchar data type and all nullable fields. Fastload expects to read a 2-byte integer at the beginning of a varchar field that contains the length of the varchar data. All nullable fields must be terminated with a single byte null indicator that determines whether the field is null.
- Fastload supports all standard table structures when loading into empty tables. It can also load data into heap tables that already contain rows.

  All other table types that contain data require a data sort that merges the loaded data with the existing data. Fastload does not perform this function. The data always loads fastest into a heap table with no secondary indexes.
- Fastload does not support complex data types such as intlist, ord_pair, or udts.
- Binary format files cannot be transported between byte-swap and non-byte swap machines. The data can be generated programmatically, but you should be careful to generate the correct record format, taking into account additional bytes needed for some field types.

Perform a Fastload Operation

To perform a fastload operation (load binary format files into a table), perform the following steps:

1. Make a backup of the table’s content.

   You need a backup because it can be difficult to fix or eradicate loaded data that is incorrect.

   Always check manually that the data has been loaded correctly.
2. Generate a copy of the file you want to fastload by copying it from an Ingres table that has the same format as the target table, or by creating it programmatically.

   Do the copy in binary format, for example:

   ```
   copy test() into 'test.out'
   ```
3. Enter the fastload command, for example:

   ```
   fastload fload -file=test.out -table=test
   ```

   The table “test” in the database “fload” is loaded from the file “test.out”.
4. Observe that a summary of the load displays the row size, number of rows loaded, and the number of bytes read.
5. Verify manually that the data has been loaded correctly.

Data Loading in a Multi-CPU Environment

It can be faster to use COPY instead of fastload if the load is being done in a multi-CPU environment. Copy is faster because it uses two processes and can use two CPUs, whereas fastload uses only one CPU. Use COPY if a large amount of sorting is required to load the data. If there is more than one CPU available on the system, fastload can become CPU-bound on a single CPU.

Advanced Use of the Copy Statement

You can perform advanced functions with the COPY statement using variations of the statement.

Examples in this section use a database that looks like the following:

Table Name	Column Name	Data Type
Header	Orderno Date Suppno Status	integer2 date integer2 char(1)
Suppinfo	Suppno Suppinfo	integer2 char(35)
Detail	Orderno Invno Quan	integer2 integer2 integer2
Iteminfo	Invno Descript	integer2 char(20)
Priceinfo	Invno Suppno Catno Price	integer2 integer2 integer2 money

Populate Multiple Database Tables Using Multiple FIles

Suppose that the information for the database previously described was stored in data files outside Ingres, and that those files, “file1” and “file2,” have the record formats shown below:

```
orderno,date,suppno,suppinfo,status
```

```
orderno,invno,catno,descript,price,quan
```

The COPY statement can be used to load the data from these files into a five-table database. Assume that the files are entirely in ASCII character format, with fields of varying length terminated by commas, except for the last field, which is terminated by a newline.

The following COPY statement loads the header table from the first file:

```
copy table header
    (orderno = char(0)comma,
    date = char(0)comma,
    suppno = char(0)comma,
    dummy = d0comma,
    status = char(0)nl)
    from 'file1';
```

Each column of the header table is copied from a variable-length character field in the file. All columns except the last are delimited by a comma; the last column is delimited by a newline.

Specification of the delimiter, although included in the statement, is not needed because the COPY statement looks for the first comma, tab, or newline as the field delimiter by default.

The notation d0 used in place of char(0) tells the COPY statement to ignore the variable-length field in that position in the file, rather than copying it. COPY ignores the column name (in this case dummy) associated with the field described as d format.

Load a Table from Multiple Files

Loading the priceinfo table presents special difficulties. The COPY statement can read only one file at a time, but the data needed to load the table resides in two files.

The solution to this kind of problem varies with the file and table designs in any particular situation. In general, a good solution is to copy from the file containing most of the data into a temporary table containing as many columns of information as needed to complete the rows of the final table.

To load data from the files into the priceinfo table, do the following:

1. Create a temporary table named pricetemp that contains the orderno column, in addition to all the columns of the priceinfo table:

   ```
   create table pricetemp (orderno integer2,
       invno integer2,
       suppno integer2,
       catno integer2,
       price money);
   ```

   Adding the orderno column to the temporary table is that it enables you to join the temporary table to the header table to get the supplier number for each row.

2. Copy the data from file2 into the pricetemp table:

   ```
   copy table pricetemp (orderno = char(0), invno =
       char(0), catno = char(0), dummy = d0, price =
       char(0), dummy = d0) from 'file2';
   ```

3. Insert into the priceinfo table all rows that result from joining the pricetemp table to the header table. (In VDBA, use an SQL Scratchpad window to execute the statement.)

   ```
   insert into priceinfo (invno, suppno, catno,price)
       select p.invno, h.suppno, p.catno,
       p.price from header h, pricetemp p
       where p.orderno = h.orderno;
   ```

4. Destroy the temporary table pricetemp:

   ```
   drop pricetemp;
   ```

Multi-line File Records

Another feature of the COPY statement is that it can read multi-line records from a file into a single row in a table. For instance, suppose that for viewing convenience, the detail file is formatted so that each record requires three lines. That file looks like this:

```
1, 5173
10179A, No.2 Rainbow Pencils
0.29
1, 5175
73122Z, 1998 Rainbow Calendars
4.90
```

Load these values into the pricetemp table with the following COPY statement:

```
copy table pricetemp (orderno = char(0)comma, invno = char(0)
    nl, catno = char(0)comma, descript = d0nl,  price =
    char(0)nl) from 'file2';
```

It does not matter that newlines have been substituted for commas as delimiters within each record. The only requirement is that the data fields be uniform in number and order, the same as for single-line records.

Load Fixed-Length and Binary Records

The COPY statement can also load data from fixed-length records without any delimiters in or between the data. In addition, numeric items in the file can be stored in true binary format. For example, the value 256 can be stored in a 2-byte integer instead of 3 characters. The order header file has the following record layout:

```
orderno date suppno suppinfo status
```

The data type formats for each of the fields is as follows:

(2-byte int) (8 chars) (2-byte int)
(35 chars) (1 char)

In this case, you code the COPY statement to load the header table as follows:

```
copy table header (orderno = integer2,
    date = char(8),
    suppno = integer2,
    dummy = char(35),
    status = char(1))
    from 'file1';
```

It is also possible to copy data from files that contain both fixed and variable-length fields.

Considerations When Loading Large Objects

Large objects are long varchar and long byte data types. Long varchar is a character data type, and long byte is a binary data type with a maximum length of 2 GB.

There are additional considerations when copying large objects into a table.

Considerations for Copying Formatted Large Objects

A column with large objects is specified for copying with the formats:

long varchar(0)
long byte(0)
long nvarchar(0)

Note: You cannot use a length specifier or a delimiter.

To handle the large size, copy deals with these data types in a similar manner as the data handlers: the data is broken up into segments for copying to a data file.

Each segment consists of the length, followed by a space delimiter, followed by the data. There is no space following a data segment (because copy knows how many bytes of data to read).

The basic structure of a formatted segment is:

```
integer        = length of segment
space          = delimiter
char|byte(len) = data
```

The last segment, or an empty object, is denoted by a zero length, followed by its space delimiter:

```
0         = length of segment
space     = delimiter
```

Thus, the data is segmented as:

```
length1 segment1  segment1length2  segment2...lengthn  segmentn0
               ^                ^          ^        ^        ^  ^
             space            space      space    space    space space
```

The segments of the long nvarchar are UTF-8 transformation of Unicode values.

For formatted copies on large object data that contain nulls, the WITH NULL clause must be specified with a value.

Example: Copying Formatted Large Objects

Consider the sample table, big_table, that was created with the following CREATE TABLE statement:

```
create table big_table
    object_id integer,
    big_col long varchar);
```

This table can be copied to the big_file file with the following COPY statement:

```
copy table big_table (object_id integer, big_col long varchar) into 'big_file';
```

Considerations for Binary Copying a Large Object

The data file format is slightly different when you copy a large object in a binary copy (that is, without column specifications).

The binary file has an extra character after the end of the last segment of a nullable column. (A nullable column is one that was created with null). The length is not followed by a space character. The basic structure of a binary segment is:

```
integer2         = length of segment
char|byte(len)   = data
```

The last segment, or an empty object, is denoted by a zero length, followed (if the column is nullable) by a character indicating whether the column is null (=0) or not null (=1):

```
0         = length of segment
[char(1) = 0 column is not null
           1 column is null]
```

Thus, a non-nullable column is segmented as:

```
length1 segment1 segment1 length2 segment2...lengthn segmentn0
```

A nullable column is segmented as:

```
length1 segment1 segment1 length2 segment2...lengthn segmentn0 0
                                                            ^
                                                        1 character
```

Empty and null strings appear as follows:

- A non-nullable empty string consists solely of the zero integer length.
- A nullable empty string consists solely of the end zeros: the zero integer length and the zero “not null” character indicator.
- A null indicator consists of “01”: an end zero integer length and the null character indicator of “1.”

Large Data Loads with the Set Nologging Statement

The SET NOLOGGING statement allows you to bypass the logging and recovery system. This can be time-efficient for certain types of batch update operations but must always be used with extreme care.

The SET NOLOGGING statement is intended to be used solely for large database load operations for which the reduction of logging overhead and log file space usage outweigh the benefits of having the system recover automatically from update errors.

Suspend Transaction Logging

To suspend transaction logging for the current session, issue the following statement:

```
set nologging
```

This statement can be issued only by the DBA of the database on which the session is operating and cannot be issued while currently executing a multi-statement transaction.

Effects of the Set Nologging Statement

After the SET NOLOGGING statement is issued, updates performed by the current session are not recorded in the log file or journal files. Updates are not reapplied if the database is rolled forward from a checkpoint, and updates do not appear in an audit trail.

When transaction logging is suspended:

- Any error (including interrupts, deadlock, lock timeout, and force-abort, as well as any attempt to roll back a transaction) results in an inconsistent database.
- The rollforwarddb operation from journals is disabled until the next checkpoint.

Before Using the Set Nologging Statement

To use the SET NOLOGGING option, you as the DBA must:

- Obtain exclusive access on the database to ensure that no updates (other than those performed by the batch update operation) can occur during the operation.
- Prepare to recover the database before suspending logging. There are two cases:
  - For existing databases, checkpoint the database prior to executing the operations that use the SET NOLOGGING statement. If an error halts processing, the database can be restored from the checkpoint and the process restarted.
  - If loading a new database, no checkpoint is required. You can handle consistency errors by destroying the inconsistent database, creating a new database, and restarting the load operation.

Important! Do not use the SET NOLOGGING statement in an attempt to improve performance during everyday use of a production database. Because the recovery procedures for failed nologging transactions are non-automated and require full database restoration, you must consider other methods if database load performance needs improving. For assistance, see the chapter “Improving Database and Query Performance.”

Restore Transaction Logging

To resume logging, issue the following statement:

```
set logging
```

The SET LOGGING statement re-enables logging for a session for which the SET NOLOGGING statement was issued.

After SET LOGGING is executed, automatic database recovery is again guaranteed.

If you use SET NOLOGGING on a journaled database, take a new checkpoint immediately after completion of the SET LOGGING operations to establish a new base from which to journal updates.

When a session operation in SET NOLOGGING mode disconnects from a database, a SET LOGGING operation is implicitly executed to bring the database to a guaranteed consistent state before completing the disconnect.

Example: Use a Set Nologging Application to Load a New Database

Here is an example sequence of using a SET NOLOGGING application to load a new database:

1. Create the database.
2. Start the program to load the new database with data. The program includes a SET NOLOGGING statement to bypass transaction logging during the data load.
3. If any errors are encountered, destroy the database and repeat Steps 1 and 2.
4. Issue a SET LOGGING statement to resume normal operations.
5. Checkpoint the database and enable journaling to enable rollforward recovery on this database.

Example: Use a Set Nologging Application to Load an Existing Database

The following sequence uses a SET NOLOGGING application to load data to an existing database:

1. Checkpoint the database and disable journaling by entering the following command at the operating system prompt:

   ```
   ckpdb -j dbname
   ```

   (In VDBA, use the Checkpoint dialog.)
2. Start the program to load the database with the new data. The program does the following:
   - Locks the database exclusively to prevent other applications from using the database until the load is complete.
   - Includes a SET NOLOGGING statement to bypass transaction logging during the data load.
3. If any errors are encountered, restore the database from the checkpoint (you can use the Database Rollforward DB menu in VDBA) and repeat Step 2.
4. Issue a SET LOGGING statement to resume normal logging operations.
5. Turn journaling back on for the database by checkpointing the database:

   ```
   ckpdb +j dbname
   ```

   This establishes a new point from which rollforwarddb processing can be done.

The load is complete. The database can be made accessible to other applications.

Successful Use of the Copy Statement

When using the COPY statement, you should avoid common problems and learn to use the statement correctly. Specifically, you should understand how to do the following:

- Check integrity errors
- Avoid reloading problems
- Control error handling
- Troubleshoot data loading

How to Check for Integrity Errors

When you use the COPY statement, the data being copied is not checked for integrity errors.

To check the integrity of your data before using the COPY statement, follow these steps:

1. Use the CREATE INTEGRITY statement (or the equivalent feature in VDBA) to impose the integrity constraint. For example:

   ```
   create integrity on personnel
       is name like '\[A-Z\]%' escape '\';
   ```

   If the search condition is not true for every row in the table, an error message is returned and the integrity constraint is rejected.

2. If the integrity constraint is rejected, find the incorrect rows; for example:

   ```
   select name from personnel
       where name not like '\[A-Z\]%' escape '\';
   ```

3. Use Query-By-Forms to quickly scan the table and correct the errors.
4. After ensuring that the data is correct, use the COPY statement to load or unload your table.

As an additional check that the information was copied correctly, apply the integrity constraint after copying the table.

For more information on integrity checking and integrity constraints, see the chapter "Ensuring Data Integrity."

Reloading Problems

When using the COPY FROM statement, the following problems in the copy file are the most frequent causes for error messages:

- Invalid data
- Miscounting fixed-length field widths
- Neglecting the nl delimiter in the COPY statement
- Omitting delimiters between fields
- Including too many delimiters

Invalid Data in the Copy File

If you try to load invalid data into a field, the row is rejected.

For example, the following record is rejected because February has only twenty-eight or twenty-nine days:

```
559-58-2543,31-feb-1998,Weir,100000.00,Executive
```

Miscounted Fixed-Length Field Widths in the Copy File

If the widths of fixed-length fields are not correct, the COPY statement can try to include data in a field that it cannot convert to the appropriate format.

For example, you receive an error message if you try to copy this row:

```
554-39-2699 01-oct-1998 Quinn 28000.00 Assistant
```

with the following COPY statement:

```
copy table personnel (ssno = char(20),
    birthdate = char(11),
    name = char(11),
    salary = char(9),
    title = char(0)nl)
    from 'pers.data';
```

Because you specified char(20), or 20-character positions, for the ssno field, the COPY statement includes part of the birth date in the value for the ssno field. When the COPY statement tries to read the birth date, it reads "998 Quinn 2" which is not a valid date if birth date is defined as a date field; if defined as a char field, you get an "unexpected EOF" error.

No nl Delimiter in the Copy File

When using fixed-length specifications in the COPY statement, you must account for the “nl” (newline) character at the end of the record.

For example, you receive an error message if you try to copy these records:

```
554-39-2699 01-oct-1998 Quinn 28000.00 Programmer
335-12-1452 23-jun-1998 Smith 79000.00 Sr Analyst
```

with the following COPY statement:

```
copy table personnel (ssno = char(12),
    birthdate = char(12),
    name = char(6),
    salary = char(9),
    title = char(10))
    from 'pers.data';
```

The format specified for the title field is char(10), which does not account for the newline character. The newline characters are converted to blanks, and the extra characters force the COPY statement to begin reading a third record that ends abnormally with an unexpected end of file. Use char(10)nl to avoid this problem.

Omitted Delimiters Between Fields in the Copy File

If you omit delimiters between fields in the data file, the record is rejected.

For example, the first record below has no delimiter between the employee’s name and her salary:

```
123-45-6789,01-jan-1998,Garcia33000.00,Programmer246-80-1357,02-jan-
1998,Smith,43000.00,Coder
```

If you try to copy these records with the following COPY statement, you receive an error message because the COPY statement attempts to read “Programmer” into the “salary” field:

```
copy table personnel
    (ssno = char(0),
    birthdate = char(0),
    name = char(0),
    salary = char(0),
    title = char(0)nl)
    from 'pers.data';
```

Too Many Delimiters in the Copy File

Be careful not to include too many delimiters in the data file. This mistake frequently occurs when you use the comma as a delimiter and it also appears in the data.

For example, in the first row, the salary value contains a comma:

```
123-45-6789,01-jan-1998,Garcia,33,000.00,Programmer

246-80-1357,02-jan-1998,Smith,43000.00,Coder
```

If you try to copy these records with the following COPY statement, you receive an error message:

```
copy table personnel
    (ssno = char(0),
    birthdate = char(0),
    name = char(0),
    salary = char(0),
    title = char(0))
    from 'pers.data';
```

You receive an error because the COPY statement reads:

- “33” as the “salary”
- “000.00” as the “title”
- “Programmer” as the next “ssno”

It attempts to read “246-80-1357” as the birthdate, which produces the error.

If you specified “title = char(0)nl”, the COPY statement still reads “33” as the salary, but it reads “000.00,Programmer” as the title. This is because it looks for a newline rather than a delimiter at the end of the title. It reads the next row correctly. Although an error message is not generated, the title field for one row is incorrect.

Error Handling with the Copy Statement

When using the COPY statement, use the various options on the WITH clause to control how invalid data is handled.

Stop or Continue the Copy

Use the WITH ON_ERROR clause to stop or continue copying the data when an error occurs. In the following example, the copy continues after finding an error:

```
copy table personnel
    (name= char(0),
    dept = char(0)nl)
    from 'pers.data'
    with on_error = continue;
```

The default is to terminate at the first error.

Stop the Copy After a Specified Number of Errors

To stop the copy after a certain number of errors, specify an error count with the ERROR_COUNT=*n* clause. For example:

```
copy table personnel
    (name = char(0),
    dept = char(0)nl)
    from 'pers.data'
    with error_count = 10;
```

The default ERROR_COUNT is 1.

This clause is not meaningful when used with the ON_ERROR=CONTINUE clause. See the Error_ Count Option for the COPY statement in the *SQL Reference Guide*.

Roll Back Rows

By default, copying data stops after finding an error. If you do not want to back out the rows already copied, specify with WITH ROLLBACK=DISABLED. For example:

```
copy table personnel
    (name = char(0),
    dept = char(0)nl)
    from 'pers.data'
    with rollback = disabled;
```

Use the WITH ROLLBACK clause on the COPY FROM statement only. Rows are never backed out of the copy file if copy into is terminated. For more information, see the *SQL Reference Guide*.

Log Errors During Copy

Use the WITH LOG clause to put invalid rows into a log file for future analysis. The following query is terminated after ten errors, and these errors are placed in a log file named badrows.data:

```
copy table personnel
    (name = char(0),
    dept = char(0)nl)
    from 'pers.data'
    with error_count = 10,
    log = 'badrows.data';
```

Continue the Copy and Log Errors

By using both LOG and ON_ERROR = CONTINUE in the clause, put invalid rows in a log file and continue to process valid ones. Correct the rows in the log file and load them into the database. For example:

```
copy table personnel
    (name = char(0),
    salary = char(0)nl)
    from 'pers.data'
    with on_error = continue,
    log = 'badrows.data';
```

Troubleshooting Tips for Data Loading

Follow these tips if you have trouble loading your data into the designated tables:

- Try loading two rows from the data file to the table until you succeed. Check the database table to be sure the results are accurate and copy the entire file.
- Use the COPY statement options to continue on error and log records that fail; examine the records later. Details are described in Control Error Handling with the Copy Statement (see page 104).

If you are not able to load data from binary files:

- Make sure that the data comes from exactly the same machine architecture. Integer and floating point formats can differ between machines.
- Pick apart your data column by column, using dummy delimiters for the rest of the row until the COPY statement succeeds.
- If all else fails, get an ASCII copy of the data so you can correct errors.

Chapter 5: Loading and Unloading Databases

This section contains the following topics:

This chapter provides information on how to unload and reload a database or selected tables using the unload database and copy database operations of Ingres. It also describes the genxml and xmlimport utilities, which are used to convert and transfer data in XML.

Unload and Copy Operations

The unload database and copy database operations are most often used to copy or move a database or selected tables from one instance to another. They allow you to copy or move data from one instance to another instance with the same or different hardware or operating system.

You can also use these commands to:

- Copy a database or tables from one database to another on the same instance
- Document your database or specific tables using the “create” scripts produced by these operations
- Make static copies of your database or selected tables for the purpose of recovery
- Archive data that you want to purge from the database or reload later

The unload database and copy database operations generate scripts that enable you to:

- Unload an entire database to external binary or ASCII files
- Copy selected tables, or all the tables, views, and procedures that you own to external binary or ASCII files
- Reload the database or objects from these files
- Export table data into XML format using the genxml utility
- Import XML data files into Ingres, using the xmlimport utility

Both the unload database and copy database operations are two-phase operations, as follows:

1. Create a script to unload or copy the table or database.
2. Execute the script to copy data out of a database and into another database.

Privilege Required for Unload Operation

To unload a database, you must be the DBA for the database or a privileged user impersonating the DBA.

Privilege Required for Copy Operation

To copy a database, you can be any user to copy selected tables or all the tables, views, and procedures that you own in the database.

Unload Operation

The unload database operation allows you to completely unload a database and reload it. You can unload an entire database or merely the objects owned by a particular user.

The unload database operation destroys the extended system catalogs and recreates them before loading the data. This is done to guarantee that the data is loaded into system catalogs identical to the ones from which they were unloaded.

Objects That Are Unloaded

The unload database operation unloads all of the objects and system catalogs in your database, including:

- Tables
- Views
- Database procedures
- Forms
- Reports
- Graphs
- Application-By-Forms definitions
- JoinDefs
- QBFNames
- Associated permissions, integrities, and indexes
- Rules
- Dbevents
- Comments
- Synonyms

When iidbdb (the Ingres master database) is the database being unloaded, the following objects are also included:

- Groups
- Roles
- Database-level privileges

Ways to Perform the Unload Database Operation

You can unload and reload a database using system commands or VDBA.

At the command line, use the unloaddb and sql commands. For details, see the *Command Reference Guide*.

In VDBA, start by using the Generate unload.ing and reload.ing dialog. This dialog is invoked by selecting a database and choosing the Database menu, Generate Scripts, and Unloaddb. For the detailed steps, see VDBA online help.

Options on the Unload Database Operation

Some of the options that are available for unloading and reloading are:

- Create printable data files
- Directory name
- Source directory
- Destination directory

Files Created During the Unload Database Operation

When you unload a database, several files are created. To ensure compatibility across all systems, the names of the generated files are truncated to twelve characters.

The generated files are as follows:

unload

Contains operating system commands to invoke a terminal monitor and execute the copy.out script

copy.out

Contains COPY statements to copy out system catalogs and all user objects

reload

Contains operating system commands to invoke a terminal monitor and execute the copy.in script

copy.in

Contains statements to destroy, create and copy in system catalogs and all user objects

The unload and reload command files have the .bat extension on Windows systems and the .ing extension on UNIX and VMS systems.

Unload in ASCII or Binary Format

When unloading a database, you should unload the files in ASCII format unless you are copying the database to another instance on the same machine or to a binary-compatible machine. In these cases, use binary format.

If you are not sure, use ASCII format.

Unloading in ASCII format allows you to:

- Move databases to an instance with a different machine architecture
- Edit the data files before reloading them into a database

To unload the database files in ASCII format, specify the -c option (create printable data files) on the copydb command. (In VDBA, use the Create Printable Data Files option in the Generate copy.in and copy.out dialog.)

To unload the database files in binary format, do **not** specify the -c option.

Note: The -c option can affect the value of floating point numbers. More information can be found in Floating Point Specification for Unload (see page 111).

Note: Copying between releases of Ingres with different major release identifiers can cause problems if new columns were added to a later release to support new features. If you have made use of these new features in the later release and attempt to unload and reload into an earlier release that did not support the new feature, the reload produces an error. A simple editing of the reload scripts to avoid loading the non-existent columns avoids this problem.

Caution! If you unload the files in binary format, do not edit them. Editing prevents you from reloading them.

Floating Point Specification for Unload

In the unload and reload command files, the floating point specification defaults to maximum precision and length (-f8F79.38).

To reduce precision or length, edit the floating point specification in these files. If you do not, zeros with no significance can consume disk space in the external data files. If overflow occurs, you can specify another flag for the output format, for example, N instead of F in the floating point specification.

Precision of formatted character output of floating point numbers is also controlled with the -f flag of the sql command. For details, see the *Command Reference Guide*.

Unload to Another Instance

When you unload a database with the Destination Directory and/or Source Directory options specified in the Generate unload.ing and reload.ing dialog, direct where the data is copied to and from. This can be on the same machine or a different machine.

When you run the unload command file, the copy.out script is executed. The copy.out script generates the data files in the destination directory. If you have specified a source directory, you must move the copy.in script and the data files to this directory. When you run the copy.in script, the user objects are created and the tables are populated with the data from the source directory.

Locking While Unloading a Database

When you perform the unload database operation or execute the unload command file, the locking system takes shared locks on the system catalogs and tables being unloaded.

When you execute the reload command file, the locking system takes exclusive locks on the system catalogs and user tables being reloaded.

Inconsistent Database During an Unload

There are two major ways that a database can become inconsistent during the unloading of a database:

- By default the database is not exclusively locked while the unload database scripts are being created or the unload command file is running. Because of this default, a user can alter tables that are not locked during this time.
- A user can alter the database after you have created the unload database scripts but before you have executed the unload command file.

  If a user drops a table in this interval, it generates an error message. However, if a user makes either of the following changes during this time, no error message is generated, and you do not know about the change:

  - Adds or deletes rows from a table
  - Adds a table

To ensure the consistency of the database while it is being unloaded, lock it exclusively.

Lock Database Exclusively During Unload

To lock the database exclusively during an unload operation, edit the unload script and add the sql command -l flag to the script, before running the unload command file.

Doing this ensures the consistency of the database.

Copy Operation

The copy database operation enables a DBA or non-DBA to copy the following:

- Selected tables and views in a database
- All of the user objects, including tables, views, and procedures, that you own in a database

Ways to Perform the Copy Database Operation

To perform the copy database operation at the command line, use the copydb command. For details, see the *Command Reference Guide*.

In VDBA, you use the Generate copy.in and copy.out dialog. This dialog is invoked by selecting a database and choosing the Database menu, Generate Scripts, and Copydb. For detailed steps, see VDBA online help.

Options on the Copy Database Operation

Some of the options that are available for copying a database are:

- Specify tables
- Create printable data files
- Directory name
- Source directory
- Destination directory

Objects that Are Copied

The database objects copied in the copy database operation depend on whether tables are specified for the operation.

The following table shows what is copied in each situation:

Options Specified	What Is Copied
No options specified.	All tables, views, and procedures (owned by the user who performed the copy database operation) and associated indexes, integrities, events, permissions, and rules.
Table/views specified.	Specified tables or views (owned by the user who performed the copy database operation) and associated indexes, integrities, and permissions.

For further flexibility in the statements written to the copy.in script, you can use additional flags so that the generated scripts contain statements to manipulate only certain database objects. The flags can be used with the specified tables or views to print statements for any particular table or view.

For example, use the –with_index flag to print statements only related to index.

For more information on these flags, see the copydb command description in the *Command Reference Guide*.

Scripts Produced by the Copy Database Operation

When you perform the copy database operation, the following two scripts are produced:

copy.out

The copy.out script contains query language statements to copy your tables to operating system files. The script contains a copy statement for each table being copied

copy.in

The copy.in script contains query language statements to recreate your tables, views, procedures, and associated indexes, permissions, and integrities, and copy the table's data from the operating system files into a database.

To copy the tables out of the database, you run the copy.out script. To copy them into the same or another database, you run the copy.in script.

If you specify a particular table or view, the copy.in script contains statements to recreate the specified table or view only (along with applicable permissions and so on). The script does not contain statements to create all tables, views, and procedures.

Ingres tables can also be copied into XML format, as described in Generate and Import XML Operations (see page 125).

Reloading Order

When using the copy.in script, database objects are reloaded in the following order:

- Users, groups, and roles (only when recreating the iidbdb)
- Tables

  ALTER TABLE statements are used as needed for deferred creation of referential integrities.
- Data

  The unload database operation (using the unloaddb command) handles all data types, including decimal data, large objects, and User Data Types (UDTs). More information can be found in Considerations When Loading Large Objects (see page 96) and Column Name and Format Specifications (see page 76).
- Table permissions

  Permissions are recreated to the original time stamp order, and can or cannot be those of the table owner (depending on the grant options for the table).
- Indexes, index modifications, and integrities
- Views and related permissions

  Like tables, these are recreated to the original time stamp order.
- Synonyms
- Database procedures and related permissions

  Procedures depend on tables, views, events, and synonyms. Procedures can also see other procedures. To handle reloading of procedures, two passes are made during the unload database process through the iiprocedures catalog (see page 453).
- Comments

Copy in ASCII or Binary Format

When copying a database, you should copy the files in ASCII format unless you are copying the database to another instance on the same machine or to a binary-compatible machine. In these cases, use binary format.

If you are not sure, use ASCII format.

Copying the files in ASCII format allows you to:

- Move the tables you own to an instance with a different machine architecture
- Edit the data files before copying them into a database

To copy the database files in ASCII format, specify the -c option (create printable data files) on the copydb command. (In VDBA, use the Create Printable Data Files option in the Generate copy.in and copy.out dialog.)

To copy the database files in binary format, do **not** specify the -c option.

Note: The -c option can affect the value of floating point numbers, as described in Floating Point Specification for Copy Database (see page 117).

Note: Copying between releases of Ingres with different major release identifiers can cause problems if new columns were added to a later release to support new features. If you have made use of these new features in the later release and attempt to copy out and copy in to an earlier release that did not support the new feature, the copy in operation produces an error. Additionally, new reserved words can have been added and can require renaming tables and/or columns. To avoid this problem, simply edit the copy.in script to avoid loading the non-existent columns, or renamed tables or columns.

Caution! If you copy the files in binary format, do not edit them; doing so causes problems.

Floating Point Specification for Copy Database

When you execute the sql command to run the copy.out and copy.in scripts, the floating point specification defaults to 10 positions with 3 to the right of the decimal. If your data requires more precision, change the precision mask by using the -f flag with the sql command when you run the copy.out and copy.in scripts.

For a description of the floating point (-f) flag parameters that is used with the sql command, see the *Command Reference Guide*.

Copy a Database to Another Instance

When copying a database, you can direct where the data is copied to and from by specifying a destination directory and a source directory. The directories can be on the same machine or different machines.

When you run the copy.out script, the data files are generated in the destination directory. If you have specified a source directory, you must move the copy.in script and the data files to this directory. When you run the copy.in script, the user objects are created and the tables are populated with the data from the source directory.

In VDBA, use the Destination Directory and/or Source Directory options specified in the Generate copy.in and copy.out dialog.

Locking While Copying a Database

When you create the copy database scripts or execute the copy.out script, the locking system takes shared locks on the tables being copied.

When you execute the copy.in script, the locking system takes exclusive locks on the tables being copied in.

Inconsistent Database During Copy Operation

There are two major ways that the database can become inconsistent during the creation of copy database scripts or the execution of the scripts:

- Because shared locks are taken on the tables being copied while the copy scripts are being created or copy.out is being executed, a user can alter the tables that are not locked during this time.
- A user can alter the tables being copied after you run the copy.out script, but before you have run the copy.in script.

  If a user drops a table in this interval, it generates an error message. However, if a user makes either of the following changes during this time, no error message is generated, and you do not know about the change:

  - Adds or deletes rows from a table
  - Adds a table

To ensure the consistency of the tables being copied, lock them exclusively while they are being copied.

Lock Database Exclusively When Copying

Locking ensures the consistency of the tables being copied.

To lock tables exclusively when copying them, use the sql command with -l flag when you run the copy.out script, as follows:

```
sql -l dbname <copy.out
```

Copy Individual Database Objects

Tables, forms, and other user objects can be moved or copied from one database to another by using various copying techniques.

Note: Make sure that there is a current backup of the database before performing any of the procedures. If there is a problem in moving the object, restore the original. For details, see the chapter "Performing Backup and Recovery."

To transfer a database object from one database to another, use the appropriate copy method, as described.

Command Scripts

The copy command creates scripts that do the following:

- Copy out the object from the current database
- Recreate the object and copy back the saved data into the new database

Prepare to Copy a Database Object

Before you copy a database object, check whether the user already owns an object by the same name in the new database.

In the case of a table, it must be destroyed before proceeding.

Caution! If you fail to do this, the new table cannot be successfully created, and you can potentially populate the existing table with unwanted data.

For other user objects (forms, reports, and applications) be aware that the object being moved can replace an object with the same name in the new database.

How to Copy a Database Object

Follow these basic steps to copy a database object:

1. Check for duplicate objects, as described in Prepare to Copy a Database Object (see page 119).
2. Log in as the DBA of the old database or as a privileged user who can impersonate the DBA (by using the -u flag for commands or by using the Users branch in the Virtual Nodes window in VDBA).
3. Use the relevant copy method to copy the object out of the database into an intermediate file.
4. Set the protections on the file copy.in or intermediate file and the data files so that you can access them after you log in as the DBA of the new database.
5. Log in as the DBA of the new database.
6. Using the relevant copy method in input mode, copy the intermediate file into the new database.

   There are now two copies of the object, one in each database.
7. To remove the original object, the DBA or privileged user must use the applicable Ingres tool as the original owner (by using the -u flag for commands or by using the Users branch in the Virtual Nodes window in VDBA) and delete the copy of the object.

Copy Tables

To copy or move data between databases, copy the relevant tables from the current database into another database.

To accomplish these tasks using system commands, use the copydb and sql commands.

In VDBA, use the Generate copy.in and copy.out dialog, invoked from the Copydb command from the Database Generate Scripts submenu. The detailed steps for performing these procedures can be found in VDBA online help.

Example: Move a Table to Another Database

In this example, the DBA moves the customers table, owned by John, from the accounts database to the orders database:

1. Enter the following command at the operating system prompt:

   ```
   copydb -ujohn accounts customers
   ```

   The copy.in and copy.out scripts are generated.

2. Enter the following commands in sequence:

   ```
   sql -ujohn accounts <copy.out
   sql -ujohn orders <copy.in
   ```

   There are now two copies of the customers table: one in the accounts database and one in the orders database.

3. The DBA removes the old table by logging into the Terminal Monitor (sql) as -ujohn and issues the following statement:

   ```
   DROP TABLE customers
   ```

Copy Forms

Copy or move one or more forms from one database to another using the copyform command.

There are two forms of syntax, one without the –i flag, which copies the forms from a database to a file. The next form, using the –i flag, copies the forms from the text file into a database.

If you are copying a form that already exists in the new database and you do **not** use the -r flag, you are prompted as to whether you want to overwrite the existing form. To do so, you select Yes. If the -r flag is specified with copyform, the form is automatically overwritten.

The copyform command can also be used with -q and -j flags, for copying QBFNames and JoinDefs. For a complete description of the flags and parameters for this command, see the *Command Reference Guide*.

Example: Move Forms to Another Database

Assume the DBA wants to move two forms owned by the DBA, customers and parts, from the accounts database to the orders database. The name forms.txt is selected as the intermediate file name. The following commands perform the move:

```
copyform accounts forms.txt customers parts
copyform -i orders forms.txt
```

At this point, there are two copies of the forms, one in accounts and one in orders. To remove the old forms, enter the Visual Forms Editor and delete them.

Copy Applications

Copying or moving an application from one database to another can be done using the copyapp command.

The copyapp command syntax has two forms: copyapp out and copyapp in.

The copyapp out command copies database objects associated with a specific application from the database to a text file. These objects are entities such as forms, reports, and join definitions.

The copyapp in command copies these database objects into the desired database.

Copyapp **does** copy all of the following:

- Forms referenced in 4GL, Query-By-Forms, and report frames
- Reports referenced in report frames
- JoinDefs referenced in Query-By-Forms frames

Copyapp **does not** copy any of the following:

- Forms (compiled or non-compiled) referenced by the 4GL-call qbf command or used in embedded query language procedures
- Reports referenced by the call report, call sreport, or call rbf 4GL command
- Graphs referenced by the call graph 4GL command

For a complete description of the flags and parameters for this command, see the *Command Reference Guide*.

Copy Reports

Move single or multiple reports from one database to another by using the copyrep command to copy the reports out and the sreport command to copy them into the second database.

The sreport command overwrites existing reports with the same names as those copied. You must check whether you have any reports with the same names as the ones being copied. If you do not want to overwrite them, edit *filename*.rw and change the report names.

For a complete description of the flags and parameters for copyrep and sreport, see the *Forms-based Application Development Tools User Guide*. Also, see the copyapp command in the *Command Reference Guide*.

Example: Copy Reports to Another Database

Assume the DBA wants to copy two reports she owns, parts_restock and parts_on_order, from the inventory database to the orders database. The file name textfile.rw is selected as the temporary file name. The following command at the operating system prompt copies the reports out to a file:

```
copyrep inventory textfile.rw parts_restock parts_on_order
```

The following command at the operating system prompt copies the reports into the orders database:

```
sreport orders textfile.rw
```

Increase Object Limit on Commands

Certain utilities like copydb, copyform, repcfg, genxml and convtouni limit to 100 the number of tables or views that can be specified on the command line. If this limit is insufficient for your application, the utexe.def file found in $II_SYSTEM/ingres/files can be modified.

To increase the 100 object limit in utexe.def file

1. Back up the utexe.def file in case you want to revert to the original file.
2. Open $II_SYSTEM/ingres/files/utexe.def for editing.
3. Search for the string %100S in the parameter description list directly under the command you want to modify.
4. Change the string to %*n*S where *n* is an integer defining the new limit.
5. Save the file and test.

Ways to Copy and Relocate a Database

Database locations can be moved, for example when a disk fills or is swapped out. You can also copy an entire database. Any location in the original database can be moved to a new location in the new database.

You can accomplish this task using the relocatedb command. For details, see the *Command Reference Guide*.

To do this in VDBA, use the Duplicate Db menu. For detailed steps, see the VDBA online help.

Example: Copy a Database to a New Database

The following series of operations copies the empdata database to a new database, empdev:

1. In a Database Object Manager window in VDBA, select a database (empdata).
2. Choose Database, Duplicate Db.

   The Duplicate Database dialog appears.
3. In the New Database edit control, enter the name of the new database (empdev).
4. Click OK.

Example: Copy a Database to a New Database and Use New Locations

The following example copies the empdata database to a new database, empdev, and specifies the new locations—empdat1 and empdat2—for the existing ii_database and edata locations:

1. Follow Steps 1–3 in the Example: Copy a Database to a New Database (see page 124).
2. Enable the Reassign Location check box.

   The locations that are currently being used by the empdata database (ii_database and edata) are displayed in the Initial Location column.
3. In the New Location column, double-click on the location to be changed and select the new location from the drop-down list box that appears.

   For example, double-click on the ii_database location in the New Location column and select empdat1. Change the edata location to empdat2.
4. Click OK.

Example: Copy a Database to a New Database and Swap Contents of Locations

The following example copies the empdata database to a new database, empdev, and swaps the contents of the locations ii_database and loc1 in the new database:

1. Follow Steps 1–3 in Example: Copy a Database to a New Database (see page 124).
2. Enable the Reassign Location check box.

   The locations that are currently being used by the empdata database (ii_database and loc1) are displayed in the Initial Location column.
3. In the New Location column, double-click on the location to be changed and select the new location from the drop-down list box that appears.

   For example, double-click on the ii_database location in the New Location column and select loc1. Change the loc1 location to ii_database.
4. Click OK.

Generate XML and Import XML Operations

The genxml and xmlimport utilities let you transfer data in XML format.

XML is a cross-platform, software and hardware independent format for transmitting information across the Internet. The XML data files produced can also be processed by other XML-enabled databases and applications.

The genxml utility converts the table data, including metadata information, into XML and places it in an XML file. You can export the whole database or specific tables into XML files. The generated XML file conforms to the generic Ingres DTD.

The xmlimport utility imports the data from an XML file into an existing Ingres database. This utility parses an XML document and prints the data and scripts into files. The script can be run to upload and store the data from the XML file into the Ingres table. The XML file for upload is validated against the Ingres DTD. Only XML files that conform to the Ingres DTD can be imported into an Ingres database.

The genxml and xmlimport database operations are run as system commands. For more information, see the *Command Reference Guide*.

Related visual tools are the Import Assistant and Export Assistant, which are used for importing and exporting data in various formats, including XML.

Chapter 6: Changing Ownership of Databases and Database Objects

This section contains the following topics:

Database Ownership

Ingres supports an ownership scheme for databases, the tables that make up the database, and related database objects, including forms and reports.

The hierarchy of ownership involves the following user classes:

- The primary system administrator
- The DBA
- The end user

Each class has a different set of ownership privileges. For details, see the *Security Guide*.

Two important rules of database ownership are:

- Objects cannot be shared among users, unless they have been granted access to the objects.
- Objects cannot be shared between databases.

At times it may be necessary to change the ownership of a database or database object, for example, when staff changes occur in your organization.

How to Change Ownership of a Database Object

When changing ownership of an object, use the appropriate copy method to:

1. Copy out the object from the database into an intermediate file
2. Copy in the object under new ownership

Prepare to Change Ownership of a Database Object

Before changing ownership of a database object, take the following preparatory steps:

1. Make sure you have a current backup of the database.
2. Check whether the user already owns an object with the same name as the object whose ownership you are changing.

   If this is the case, the existing duplicate object must be destroyed before proceeding.

   Important! If you fail to do this, the new object, owned by the new user, cannot be successfully created, and you can potentially corrupt the existing object with unwanted data.

Change Ownership of a Database Object

Follow these basic steps to change ownership of a database object:

1. Take preparatory steps, as described in Prepare to Change Ownership of a Database Object (see page 128).
2. Log in as the DBA of the database.
3. Use the relevant copy method to copy the object out of the database into an intermediate file.
4. Using the relevant copy method in input mode, copy the intermediate file into the new database as the new owner (by using the Users branch in the Virtual Nodes toolbar in VDBA, or by using the -u flag for commands).

   There are now two copies of the object in the database, one owned by the original owner and one owned by the new owner.
5. To remove the original object, the DBA or privileged user uses the applicable Ingres tool as the original owner (by using the Users branch in the Virtual Nodes toolbar in VDBA, or by using the -u flag for commands) and deletes the original object.

Change Ownership of Tables

Follow these steps to change the ownership of a table:

1. Generate the executable scripts. In VDBA, use the Generate copy.in and copy.out dialog box, invoked from the Copydb command from the Database Generate Scripts submenu.
2. Execute the copy.out script, copying the table as the current owner.
3. Execute the copy.in script, copying the table back in as the new owner.

For detailed steps for performing these procedures in VDBA, see online help.

These tasks can also be accomplished using the copydb and sql commands. For more information, see the *Command Reference Guide*.

For more information on copying tables and other database objects, see the chapter "Loading and Unloading Databases."

Example: Change Ownership of Table

The following example changes the ownership of any table from the currently selected user, John, to the user named dba.

1. In VDBA, open the Generate copy.in and copy.out dialog box for the database in which the table is located. For more information, see online help.
2. Click Tables to invoke the Specify Tables dialog box. Enable the check box for the table whose ownership you want to change and click OK.
3. Click OK to create the copy scripts.
4. At the operating system prompt, enter the following command to copy the table from the database mydb into an intermediate binary file in the current directory:

   ```
   sql -ujohn mydb <copy.out
   ```

5. Edit the copy.in file to change the table reference in the GRANT statement from john.<table> to <dbaname>.<table>. If you do not do this, the GRANT statements refer to John's table.

   Note: The GRANT statements are present only if grants are defined for the table being copied.

   There are now two copies of the table, one owned by the user john and the other owned by the user dba. In the usual case, John's version is no longer needed and can be removed. For example, the user john (or another user impersonating john) can easily drop the table in VDBA. For more information, see online help.

6. At the operating system prompt, enter the following command to copy the table from the intermediate binary file in the database as user dba:

   ```
   sql -udba mydb <copy.in
   ```

You also need to grant access permissions to the new table owned by user dba. For more information, see the *Security Guide*.

Change Ownership of Applications

To change the ownership of an application (created with Applications-By-Forms or Vision) from any current owner to any new owner, use the copyapp command.

The command syntax has two forms: copyapp out and copyapp in. To change the ownership, you issue the first form of the command under the current ownership, and the second form under the new ownership.

Example: Transfer Ownership of an Application to Another User

Assume the user john wants to transfer ownership of the application named app1 in the database mydb to the user dba. The following commands, entered at the operating system prompt, accomplish this, using the default intermediate text file and the current working directory:

```
copyapp out mydb app1 -ujohn
copyapp in mydb iicopyapp.tmp -a -udba
```

At this point, there are two copies of the application, one owned by the user john and the other owned by the user dba. In the usual case, John’s application is no longer needed and can be removed using Vision or Application-By-Forms.

For a complete description of the flags and parameters for this command, see the copyapp entry in the *Command Reference Guide*.

Change Ownership of Forms

Change the ownership of a form from any current owner to any new owner using the copyform command. There are two forms of syntax, one without the –i flag, which copies the forms from a database to a file. The next form, using the –i flag, copies the forms from the text file into a database. To change the ownership, issue the first form of the command under the current ownership, and the second form under the new ownership, as shown in the example that follows.

Example: Transfer Ownership of Forms to Another User

Assume the user john wants to transfer ownership of the forms named customers and parts in the database mydb to the user dba. The following commands, entered at the operating system prompt, accomplish this:

```
copyform -ujohn mydb forms.txt customers parts
copyform -i -udba mydb forms.txt
```

For a complete description of the flags and parameters for this command, see the copyform entry in the *Command Reference Guide*.

At this point, there are two copies of each form, one owned by the user john and the other owned by the user dba. In the usual case, John's forms are no longer needed and can be removed in Visual Forms Editor.

Change Ownership of Reports

Change the ownership of a report from any current owner to any new owner using the copyrep and sreport commands. The copyrep command copies the reports out under the current ownership, and sreport copies them back in under the new ownership, as shown in the example that follows.

Example: Transfer Ownership of Reports to Another User

Assume the user john wants to transfer ownership of the reports named parts_restock and parts_on_order in the database mydb to the user dba. The following commands, entered at the operating system prompt, accomplish this:

```
copyrep -ujohn -f  mydb text.rw parts_restock parts_on_order
sreport -udba mydb text.rw
```

For a complete description of the flags and parameters for copyrep and sreport, see the *Forms-based Application Development Tools User Guide*. These commands are also described in the *Command Reference Guide*.

At this point, there are two copies of each report, one owned by the user john and the other owned by the user dba. In the usual case, John's reports are no longer needed and can be removed in Report-By-Forms.

How to Change Ownership of a Database

At times, you may need to change the ownership of an entire database, for example, when a database moves from development to production or when the current DBA moves to a different project.

To change the ownership of a database, you must have permission to impersonate another user and to update system catalogs.

To change the ownership of a database, follow this process:

Note: In this process, the user name of the current owner is user_old and user name of the new owner is user_new.

1. Be sure that there is a current backup of the database, preferably a checkpoint. For more information, see the chapter “Performing Backup and Recovery.” If there is a problem in changing ownership, restore the original database.
2. Log in as the current DBA of the database.
3. Create a temporary working directory to hold the files that can be created. Move to that directory. Be certain that the temporary directory is not in the path pointed to by ING_ABFDIR or you will lose your unloaded files during destroydb.
4. Create the unload and reload scripts using VDBA.

   Note: If you are also moving the database to a machine with a different processor you must unload the database with the Create Printable Data Files option enabled. Doing so produces data files in a portable, ASCII format.
5. Unload the database by executing the unload script at the operating system prompt. The name of this file is described in Files Created During the Unload Database Operation (see page 110).
6. On UNIX, change permissions, as follows, so the new database owner can work with these files:

   ```
   chmod 744 *
   ```
7. Destroy the original database by dropping it from within VDBA. For more information, see online help.
8. Log in as the new database owner or impersonate the new owner by selecting the appropriate user name from the Users branch in the Virtual Nodes toolbar in VDBA.
9. Create a fresh database in VDBA, which can be owned by the user chosen is Step 8. For details, see online help.

10. Log in as the installation owner and go to the directory containing the reload script created in Step 4. The name of this file is described in Files Created During the Unload Database Operation (see page 110).

    The reload script contains a line for each user who owns objects (tables, indexes, or views).

11. Edit the reload script:

    a. Change those lines that reload objects with the user flag of the old owner, so that they can load with the user flag of the new owner.

    b. Take ownership of the database objects of any or all users by changing each user line so that it loads with the new user flag.

    Caution! The user flag for user $ingres must never be changed. $ingres is a special user ID that is used internally for the system catalogs.

12. Reload the database by executing the reload script. For more information, see online help.

13. Run system modification to update the query optimizer information. For more information, see the chapter "Using the Query Optimizer."

    At this point all objects (including tables, indexes, and views) are owned by the new DBA; however, database objects (forms, reports, applications, and so on) need special attention to make them accessible to everyone, because they are still owned by their old owners.

14. Update the ii_objects catalog to change ownership of these objects to the new DBA:

    a. Make sure that the new DBA does not already own any objects (forms, reports, and so on) with names identical to those you are about to reassign. If there are two identically named objects for the same owner, the original is overwritten and destroyed.

       Run the following query to select the database objects for the old owner, user_old:

       ```
       select object_id, object_owner
         from ii_objects
         where object_owner = 'user_old';
       ```

       Run the following query to select the database objects for the new owner, user_new:

       ```
       select object_id, object_owner
         from ii_objects
         where object_owner = 'user_new';
       ```

    b. Compare the object list for the new owner with the list for the new owner. If duplicates are found, eliminate them by deleting or copying and renaming the objects.

    c. After you have copied and renamed or destroyed any duplicates, rerun the queries to ensure that there are no longer any duplicate objects.

15. Execute the following query to transfer ownership of existing database objects, for example from the VDBA SQL Scratchpad window:

    ```
    update ii_objects set object_owner = 'user_new'
      where object_owner = 'user_old';
    ```

    Note: You can execute this query from a terminal monitor only if you invoke it using the +U flag, which allows you to update the system catalogs and secondary indexes.

16. Test the database and remove the temporary working directory and the associated work files.

Chapter 7: Maintaining Databases

This section contains the following topics:

Maintaining your databases keeps them in good condition and helps you to more quickly identify any problems.

Ways to View Database Objects

The DBA must make sure important database objects, such as tables and views, are available, devise a way to separate temporary objects from important objects, and keep private objects to a minimum.

You can view database objects using the HELP statement. By using options such as INDEX, TABLE, and VIEW, you can obtain information on various types of database objects. For details, see the *SQL Reference Guide*.

In VDBA, you can view a list of database objects in the Database Object Manager window. You can view details for any object in the tree by selecting it and using the panes to the right of the tree structure. By default, when you open a Database Object Manager window, only the objects belonging to you are visible. For more information, see VDBA online help.

View Database Objects that Belong to Another User

To view and work with database objects belonging to another user, you must impersonate that user (which requires the security privilege).

To impersonate another user, select that user from the Users branch in the Virtual Nodes window in VDBA and open a Database Object Manager window. The objects belonging to that user and those belonging to the DBA appear in the window, where you can view and manage them.

List All Tables and Their Owners

The iifile_info view (see page 140) permits you to select all tables and their owners.

For example, the following query lists all user tables not owned by the DBA:

```
select tablename, table_owner, table_type
  from iitables
  where table_owner != '$INGRES' and
  table_owner != 'DBA';
```

Ways to Delete Database Objects

Database objects, such as tables, views, secondary indexes, and synonyms, can be deleted (dropped). When you drop a table, objects that are directly dependent on that table, such as indexes and views, are automatically dropped.

In SQL, you can accomplish this task using the DROP statement. For details, see the *SQL Reference Guide*.

In VDBA, you can perform these tasks in the Database Object Manager window. The online help topic Dropping Objects gives a generic description for dropping any type of database object. Each type of object has its own help topic, such as Dropping a Table or Dropping a View. If for some reason you cannot drop tables in VDBA, you can use another method. More information can be found in Verifying Databases (see page 139).

Routine Database Maintenance Tips

To keep your tables in good condition, we recommend that you run the following maintenance tools periodically:

- Modify database tables periodically if they are subject to frequent updates or inserts. Frequent updates and inserts to all table structures except B-tree cause overflow data pages to be created, which are inefficiently searched.

  Note: B-tree tables with 2K pages can develop overflow from leaf pages with highly duplicate keys; B-trees with larger pages cannot develop overflow.

  If you do not have enough disk space to modify a large B-tree table, modify the table to shrink the B-tree index. This improves the structure of the B-tree index pages, but does not require the amount of free disk space required by other modify options.

  For details on how to modify tables, see the chapter "Maintaining Storage Structures."

  Note: Choosing the correct storage structure for your needs makes maintaining the database easier. For a discussion of the four main storage structures, see the chapter "Choosing Storage Structures and Secondary Indexes." If the storage structure you are using is not the best one, modify it using the information in the chapter "Maintaining Storage Structures."

- Run system modification on the database if the database is active (that is, users frequently create or modify tables, views, or other database objects). Both system catalog data page overflow and locking contention is reduced by regular use of system modification. For details, see Example: Before and After Optimization in the chapter "Using the Query Optimizer."

- Use optimization to help maintain databases. When you optimize a database, data distribution statistics are collected that help queries run more quickly and use fewer system resources. We recommend that you optimize your database when its data distribution patterns change.

  Optimization cannot be run on all columns of all tables in your database. Instead, run it only on those columns that are commonly referenced in the WHERE clauses of queries. Collecting more statistics than you need consumes extra disk space and requires the query optimizer to consume more system resources to arrive at an appropriate query execution plan.

  For details on optimization, see Database Statistics in the chapter "Using the Query Optimizer."

Note: You can set up these routine maintenance tasks to be done inside maintenance batch jobs to avoid the need to run them interactively.

Operating System Maintenance Tips

It is important for you, as the DBA, to monitor the operating system. If you are not also the system administrator, you must work closely with your system administrator so that you are aware of any operating system problems.

Ingres relies on the operating system to access data in tables. If the operating system develops problems, such as system resource shortages, lack of free disk space, or hardware errors, this can affect the responsiveness of the Ingres system and its ability to process requests on behalf of its clients.

Disk errors, memory errors, or operating system resource shortages are the problems most likely to affect the quality of operation. Most hardware errors are dependably logged by the operating system. Make sure that the system administrator is aware of your concern about the efficiency of the operating system.

The operating system offers tools to check and verify the health of the hardware. These include disk drive verification programs and diagnostic programs for memory boards.

Windows: Windows lets the system administrator check for and optionally fix problems in a file system. Free disk space and system configuration can be monitored with the Windows Diagnostics. System-wide performance data, such as CPU usage, can be monitored using the Performance Monitor. Certain system-wide errors and events are monitored in the Event Log, which can be viewed with the Event Viewer. For information on these and other administrative tools, see the Windows documentation.

UNIX: Most UNIX vendors have a fsck program to check for unreferenced disk blocks, unreferenced inodes, and inconsistencies in operating system tables. Free disk space in your file systems is easily monitored with operating system tools such as df and du. The pstat (BSD) or sar (System V) UNIX commands have options to show the use and distribution of various operating system resources. Every vendor also provides a variety of system maintenance utilities that are menu-driven and easy to use, but which are generally specific to a particular operating system vendor. Make full use of any operating system tools such as these.

VMS: VMS offers the analyze command which, among other operations, analyzes readability and validity of files and disk volumes. The show device command shows the amount of free disk space. The VMS Monitor Utility (MONITOR) monitors classes of system-wide performance data, such as CPU usage, at a specified interval. These are only a few of the system maintenance utilities that VMS provides. Consult the VMS Help facility and your VMS System Manager for more information on these and other useful operating system tools.

Verifying Databases

The Verify Database operation lets you verify the integrity of a database and repair certain table-related problems.

You can verify one or more databases by specifying an operation, and then choosing an appropriate scope and mode for that operation. Operations include:

- Checking specified tables for inconsistencies and recommending ways to repair them
- Checking database system catalogs for inconsistencies and recommending ways to repair them
- Purging temporary tables, which can be left on the disk inadvertently when the system does not have time to shut down in an orderly fashion (for example, if the machine is rebooted or stops due to power loss)
- Purging expired tables
- Dropping tables that cannot be dropped in the normal manner (for example, if the underlying disk file for the table was deleted at the operating system level) by removing all references to them from the database system catalogs
- Checking the specified databases to determine if they can be and indicates whether the user can connect to the database accessed

In VDBA, use the Verify Database dialog. For details on how to specify an operation using the Verify Database dialog, see online help.

You can also accomplish these tasks using the verifydb system command. For more information, see the *Command Reference Guide*.

To use the verify database operation, you must be the DBA for all the databases you want to verify, or a user with the security or the operator privilege.

Databases Shared Among Multiple Users

Follow these rules for databases that are shared among multiple users:

- Have users use only application programs to access data in the database. Discourage users from using Ingres tools, such as a terminal monitor or VDBA, to access data. Permitting users to access data only by means of an application program guarantees that the executing queries were written by an application programmer and are not ad hoc queries that can damage or delete data, or cause lock contention delays.
- Ensure that reports are run with readlock=nolock (see page 325). You can do this by including all reporting tools in application programs and setting readlock there, or by running all reports from operating system scripts, which set lockmode before the report runs. Doing this avoids locking contention problems that can lead to severe concurrent performance problems in the database.

How File Names Are Assigned for Tables

A naming algorithm is used to assign underlying file names for tables. There are two columns in the iirelation table used to produce names:

- reltid, a unique table identifier assigned in sequential order
- reltidx, a unique index identifier associated with each base table

The algorithm for creating the name is as follows:

1. Convert reltid (for base tables) or reltidx (for secondary indexes) to an 8-digit hexadecimal number.
2. Assign letters to each of the resulting hexadecimal digits:

   0,1, 2, ..., F is assigned to a, b, c, ..., p

For example, a reltid of 129 converted to an 8-digit hex number is "00000081". Substituting letters gives a file name of aaaaaaib.t*nn*, where *nn*=00, 01, ..., for first (or only) location, second location, and so on.

Select File Names Associated with Tables

As the DBA, you can select the names of the disk files associated with tables by using the iifile_info view, as shown in this example:

```
select table_name, owner_name, file_name, file_ext
    from iifile_info;
```

Retain Templates of Important Tables

A good practice is to periodically generate copy scripts for important tables and views. The copy.in scripts are useful if you need to recreate new, empty tables, or the entire database.

To generate copy scripts, use the unloaddb or copydb commands, or use the Generate copy.in and copy.out dialog in VDBA.

Chapter 8: Ensuring Data Integrity

This section contains the following topics:

Data Integrity Through Integrities, Rules, and Events

The following mechanisms can be used to enforce data integrity:

- Integrities
- Rules
- Database events

You can use these mechanisms to enforce a variety of relationships—such as referential integrity and general integrity constraints—or for more general purposes, such as tracking all changes to particular tables or extending the Ingres permission system.

Data integrity in the form of constraints was introduced in the chapter "Managing Tables and Views."

Integrities

The integrity mechanism is similar to the referential, unique, check, and primary key constraints for ensuring data integrity when you create or alter a table.

Constraints Compared with Integrities

Constraints check for appropriate data values whenever data is entered in the table. For more information, see the chapter "Managing Tables and Views."

Integrity refers to integrity objects defined after the table is created to check on update requests before they are allowed to affect the database.

Both mechanisms can be used to ensure data integrity.

Note: Constraints are the ISO Entry SQL92-compliant methods for maintaining database integrity and are, therefore, recommended over integrities. We recommend that you not define both constraints and integrities in the same table.

Differences in Error Handling Between Integrities and Constraints

Constraints and integrities differ in their error-handling characteristics:

- If a constraint is defined for a table, an attempt to update the table with a row containing a value that violates the constraint causes the DBMS to abort the entire statement and issue an error.
- If an integrity is defined for a table, an attempt to update the table with a row containing a value that violates the constraint causes the invalid row to be rejected, but no error is issued.

Important! If you mix constraints and integrities in the same table, the integrities are checked first. If a row violates both an integrity and a constraint, the row is filtered out by the integrity before the constraint is checked, and thus does not generate an error message.

Differences in Null Handling Between Integrities and Constraints

Constraints and integrities handle nulls differently. Check constraints allow nulls by default, whereas integrities do not allow nulls by default. Instructions on how to allow nulls are described in Nulls and Integrities (see page 146).

Working with Integrity Objects

An integrity object defines an automatic check that allows you to closely monitor any update requests before they are allowed to affect the database.

You can perform the following basic operations on integrity objects:

- Create integrity objects
- View existing integrity objects, including the detailed properties of each individual object
- Drop integrity objects

In SQL, you can accomplish these tasks with the CREATE INTEGRITY, HELP INTEGRITY, and DROP INTEGRITY statements. For complete details, see the *SQL Reference Guide*.

In VDBA, use the Integrities branch for a particular table in the Database Object Manager window. For detailed steps, see online help.

How Integrities Are Used

Immediately after you define an integrity object, the table is checked to make sure that the condition is true for all existing rows. If not, an error is returned, and the integrity object is rejected. If your table is very big, it takes some time to scan each row to determine whether the integrity can be applied.

After successfully creating an integrity object, all subsequent operations on the table must satisfy the specified condition. Changes to the database (that is, updates, inserts, and deletes) that are not applied because of an integrity violation are not specifically flagged or reported as errors—they are simply not performed:

- If a change applies to a set of rows, this means that only some of the rows were actually updated.
- If the change is for a single row, a returned row count of zero is a clue that the update did not take place.

Nulls and Integrities

If you create an integrity involving a column that is nullable (has been created using the WITH NULLS clause so the user can insert a NULL), the condition must take into consideration the possibility of encountering a null value. For more information on nullable columns, see the chapter "Managing Tables and Views." For example, suppose the number column in a particular table is nullable, and you define an integrity with the following condition that restricts number values to 50 or less:

```
number <= 50
```

Null is not in itself a value, so the comparison evaluates to false for any row in which the number column already has a null entry. You must create this integrity on a nullable column **before** the column contains any nulls. Otherwise, the integrity is rejected. Furthermore, with this integrity defined, the number column, even though it is defined as nullable, does not allow nulls.

To to allow nulls in the column, you need to define the integrity with a NULL clause to ensure proper handling of nulls with the integrity constraints:

```
number <= 50 or number is null
```

The Copy Statement and Enforcing Integrities

If you use the COPY statement where integrities are involved, after the copy operation you must check for and replace or delete rows with values violating the integrities. Alternatively, you can copy to a temporary table and create an INSERT statement that uses a subselect statement on the temporary table.

Note: Constraints defined when you create or alter a table are also ignored in this situation and must be dealt with in a similar manner.

Rules

A *rule* is a user-defined mechanism that invokes a database procedure whenever the database changes in a specified way, for example by insert, update, or delete.

A rule is a general-purpose mechanism that can be implemented for many purposes. For example, integrities can be implemented by rules. With integrities, violations are not specifically flagged or reported as errors. With rules, however, you can control exactly what happens when a violation occurs by defining in the database procedure the actions to take.

Rules and Database Procedures

A rule is always associated with a database procedure that is executed when the rule is fired. Before creating a rule, you must create its corresponding database procedure, and you must have execute privileges for the database procedure invoked by the rule. For details, see the *Security Guide*.

Working with Rule Objects

You can perform the following basic operations on rules:

- Create rule objects
- View existing rule objects, including the detailed properties of each individual object
- Drop rule objects

In SQL, you can accomplish these tasks using the CREATE RULE statement, DROP RULE statement, and the RULES and NORULES options of the SET statement. For complete details, see the *SQL Reference Guide*.

In VDBA, use the Rules branch for a particular table in the Database Object Manager window. For the detailed steps, see the Procedures section of online help.

How Rules Are Used

After a rule object is created, the rule is stored with the table in the database and is applied continuously. Whenever the execution of a statement satisfies an existing rule condition, that rule is *fired*, meaning that the database procedure associated with the rule is executed. There is no need for application code to explicitly enforce the rule.

It is also possible for a statement in a rule-invoked database procedure to fire another rule. Rules can be nested in this manner up to a maximum level specified by the DBMS configuration parameter, rule_depth.

Any user who has the privilege to access the table through the operation specified by the rule has implicit permission to fire the rule and execute its associated database procedure. For information on privileges and how they are defined, see the *Security Guide*.

Before and After Rules

Rules can be defined to execute before or after the effect of the triggering statement is applied. AFTER rules are more common and are used to perform auditing operations, integrity checks, and other operations on the updated rows. BEFORE rules can be used to validate and replace values in an inserted or updated row before the row is stored in the database. Both types of rules can be used to inhibit the execution of the triggering statement if an error condition is encountered, although BEFORE rules can typically do so more efficiently.

Example: Use a Rule to Implement the Equivalent of an Integrity

For example, if you wanted to implement a rule equivalent to an integrity in which the condition was salary <= 50000, create the rule by filling in the VDBA Create Rule dialog as follows:

1. For Rule Name, enter **check_salary**.
2. For After, enable Insert and Update. This way, the rule is fired when new rows are added and when existing rows are updated.
3. For Specify Columns for Update, enable salary, because that is the only column you need to check after an update.For Where, enter **new.salary > 50000**.

   Here, "new" is a *correlation name*, and "new.salary" is a *correlation reference*. With any column name, you can specify whether you want to use its value before or after the update using the correlation name "old" or "new," respectively.

   Note: Unlike an integrity check, which specifies a condition that cannot be violated, a rule specifies a where condition that must be met. Thus, the integrity condition (salary <= 50000), and the rule where condition (shown in the step above), are opposites.
4. For Procedure Name, enter the name of the database procedure to execute when this rule is fired (for example, **salary_too_big**). This procedure must exist when the rule is created.
5. For Parameters, enter any parameters required by the salary_too_big procedure.

The specified database procedure is executed and sent any specified parameters when the salary column has a value that is too big (that is, greater than $50,000). The procedure can be written to reject the operation, causing it to be rolled back; log the error, but allow the operation; or modify the value so that it is less than or equal to $50,000, and log that action.

Rules and Transactions

The statement that fires a rule and the database procedure invoked by the rule are considered part of the same single query transaction. Consequently, the database procedure invoked by the rule is executed before the statement that fired the rule completes. Because of this, you cannot issue a COMMIT or ROLLBACK statement in a database procedure invoked by a rule.

If the database procedure does not exist when the rule is invoked, or if an error occurs in the execution of a rule, the response is as if the statement firing the rule has experienced a fatal error. Any changes made to the database by the statement and any made by the fired rule are rolled back.

Enforcing Referential Integrity

A *referential integrity* asserts a relationship between two tables such that the values in a column of one table must match the values in a column of the second table. Traditionally, the two tables have a *parent-child* relationship:

- The parent table has a column, called the primary key, containing values against which other values are compared. The primary key is normally unique.
- The child table has a column, called the *foreign key*, whose values must match those of the primary key in the parent table.

A primary key does not have to be referenced by a foreign key (that is, there can be a parent without a child). However, every foreign key must match a primary key. There cannot be a child without a parent (that is, an *orphan*)—this constitutes a referential integrity violation.

For example, for the parent table, create a rule to fire on an update or delete of the primary key (an insert simply creates a parent without a child, which is not an integrity violation). The database procedure can check for foreign keys that reference the primary key and enforce the referential integrity.

For example, for the child table, create a rule to fire on an update or insert of the foreign key. The database procedure checks to make sure there is a parent.

The advantage of using a rule (as opposed to a constraint) to enforce referential integrity is that the actions performed by a rule can be more complex than merely checking for the existence of a primary key in a parent table. For example, a rule can fire a procedure to create an appropriate parent record if one does not already exist.

There are a number of ways that a referential integrity violation can be handled. Three common techniques are to reject, nullify, or cascade the firing statement.

Reject Technique for Enforcing Referential Integrity

Rejecting a value that violates an integrity constraint rolls back the statement that fired the rule. The raise error statement performs this function, informing the application that the results from the statement firing the rule violated some specified condition or constraint. The response to a raise error statement is the same as if the statement that fired the rule experienced a fatal error—the firing statement is aborted and any changes to the database resulting from the statement and subsequent rule firing are rolled back.

Example: Enforce Referential Integrity Between an Employee and Manager

For example, the following database procedure can be invoked by a rule to enforce referential integrity between an employee and the employee's manager and department. The code for the procedure, which has a procedure name of valid_mgr_dept, is shown as it is entered in the VDBA Create Procedure dialog:

Parameters

```
ename varchar(30),

mname varchar(30),

dname varchar(10)
```

Declare Section

```
msg varchar(80) not null;

check_val integer;

mgr_dept varchar(10);
```

Statements

```
/* Check to see if there is a matching manager */

select count(*) into :check_val from manager

  where name = :mname and dept = :dname;
if check_val = 0 then
  msg = 'Error 1: Manager "' +
    :mname + '" not found in that dept.';
  raise error 1 :msg;
  return;
endif;

/* Check to be sure there is a matching dept */
select count(*) into :check_val
  from dept where name = :dname;
if check_val = 0 then
  msg = 'Error 2: Department "' +
    :dname + '" not found.';
  raise error 2 :msg;
  return;
endif;
msg = 'Employee "' + ename + '" updated ' +
  '(mgr = "' + mname + '", dept = "' + dname + '")';
message :msg;
insert into emplog values (:msg);
```

This procedure checks the manager table to make sure that the employee's manager manages the department to which the employee is assigned. It checks the department table to see that the department is valid. If any of these checks fail, an error is issued and the new employee is not inserted into the employee table. If the constraint is met, a message is displayed and a log record is inserted into a journal table.

After defining this database procedure, create a rule to invoke it after updates and inserts, and enter the following for the procedure parameters:

```
ename = new.name, mname = new.mgr, dname = new.dept
```

Note: Any value referring to a column name in a parameter list must be preceded by a correlation name. Using the correlation name "old" or "new," specify whether you want to use the column value before or after the operation, respectively.

Nullify Technique for Enforcing Referential Integrity

Nullifying is a second course of action in response to a violation of a referential integrity constraint if a foreign key does not have a matching primary key. (Nullifying means that the columns in the records in violation of the constraint are made null, as opposed to deleting the records or returning an error that the constraint was violated.)

You are not restricted to nullifying the foreign key. You can modify the value to another defined value. Because null is not a value, it traditionally does not participate in the referential integrity relationship. Thus, a child row with a null foreign key value is not generally considered an orphan. However, rules provide you with the facilities to do such things as simulate matches on nulls.

For example, the following database procedure, nullify_children, can be invoked by a rule, when a parent row is deleted, to nullify all child entries belonging to that parent:

Parameters

```
me varchar(10)
```

Declare Section

```
msg varchar(80) not null;
```

Statements

```
msg = 'Nullifying child(ren) of "' + :me + '"';

message :msg;

update person set parent = NULL where parent = :me;

if iirowcount > 0 then
  msg = 'Nullified ' + varchar(:iirowcount) +
    ' child(ren) from "' + :me + '"';
else
  msg = 'No children nullified from "' + :me + '"';
endif;
message :msg;
```

After defining this database procedure, create a rule to invoke it after deletes, and enter the following for the procedure parameters:

```
me = old.name
```

Cascade Technique for Enforcing Referential Integrity

Cascading is the third available option in response to a violation of a referential integrity constraint. (Cascading means that the original update applies to other records that violate the constraint.) If the statement that violates the constraint is:

- An insert or update, cascading consists of inserting the offending foreign key into the primary key column.
- A delete, cascading means not only deleting the primary key, but also deleting all foreign keys that match that primary key.

The database procedure shown in this example, delete_children, can be used to implement a cascading delete rule. The procedure can be invoked by a rule, when a parent row is deleted, to delete all child entries belonging to that parent:

Parameters

```
me varchar(10)
```

Declare Section

```
msg varchar(80) not null;
```

Statements

```
msg = 'Deleting child(ren) from "' + :me + '"';

message :msg;

delete from person where parent = :me;

if iirowcount > 0 then
  msg = 'Deleted ' + varchar(:iirowcount) +
    ' child(ren) from "' + :me + '"';
else
  msg = 'No children deleted from "' + :me + '"';
endif;
message :msg;
```

After defining this database procedure, create a rule to invoke it after deletes, and enter the following for the procedure parameters:

```
me = old.name
```

When the rule is fired after the initial delete statement, it executes the delete_children database procedure, which deletes all children whose parent is the current person. Each delete statement in the delete_children procedure, in turn, also fires the delete rule, until a particular person has no descendants. The message statements that are executed before and after a row is deleted demonstrate the order in which the tree is traversed.

Note: In this example, the person table is *self-referencing*, and functions like a self-join. Referential integrity does not require two separate tables. Here the primary key is name and the foreign key is parent, both of which are in the person table.

Enforcing General Integrities

To set up tables that maintain data calculated from other tables, use views on normalized tables. For functional, performance, or data distribution reasons, the derived data must be maintained in another table or even in a specific column of the same table.

A *general integrity* is any integrity check that is not a referential integrity. General integrities can be used, for instance, to describe the relationship between the original data and the derived data, and a rule can be used to enforce the described relationship.

For example, consider two tables, employee and department. The employee table contains employee information, including the name of the department in which each employee works. The department table includes the number of employees in each department. Given these tables, a useful general constraint is that the number of employees listed for a row in the department table must match the number of employees in the employee table who work in that department.

This constraint can be enforced using rules to correctly update a row in the department table whenever an employee is hired, leaves, or changes departments. For example, if you create a database procedure that updates the department table whenever a new employee is hired, define a rule to invoke it after an insert, passing the department number as a parameter.

Enforcing General-Purpose Rules

General-purpose rules are those rules that do not fall in the category of either referential or general integrity constraints.

Using a Rule to Apply External Resource Controls

You can use general purpose rules to apply external resource controls.

For example, if you have a table of items in stock, define a rule that fires after an update to the in_stock column. The following WHERE clause causes the rule to fire if the number of items in stock is reduced to less than a minimum value of 100:

```
items.in_stock < 100
```

The rule executes a database procedure that reorders the item responsible for firing the rule, passing as parameters an item identifier and the number of items in stock. For example:

```
id = items.id, items_left = items.in_stock
```

Using a Rule to Extend the Permission System

A rule can be created to extend the permission system by ensuring that unauthorized users cannot modify certain classified rows in the opcodes table. The rule, which must be fired after inserts and deletes, is defined with the following WHERE clause:

```
opcodes.scope = 'share' and user != 'system'
```

The database procedure invoked by this rule can issue an error (using the RAISE ERROR statement, which rejects the statement that fired the rule) and log the operation with the user name into a local log table for later review (the next example demonstrates logging).

Example: Use a General Purpose Rule to Track Changes to Personnel Numbers

This example tracks changes to personnel numbers. When an employee is removed, an entry is made into the manager table, which in turn causes an entry to be made into the director table. Even if an entry is made directly into the manager table, the director table is notified.

To implement this, two database procedures need to be defined. The first, manager_emp_track, updates the manager table by reducing the number of employees for a manager, and inserts an entry into a separate table, mgrlog, to log which employee was deleted for the manager:

Parameters

```
ename varchar(30),

mname varchar(30)
```

Statements

```
update manager set employees = employees – 1

  where name = :mname;

insert into mgrlog values ('Manager: ' +
  :mname + ', Deleted employee: ' + :ename);
```

The second, director_emp_track, updates the director table by reducing the number of employees for a director:

Parameters

```
dname varchar(30)
```

Statements

```
update director set employees = employees - 1

  where name = :dname;
```

Two rules also need to be defined. The first one, defined for the employee table, executes manager_emp_track after a delete operation, passing the following parameters:

```
ename = old.name, mname = old.manager
```

The second rule, defined for the manager table, executes director_emp_track after an update operation on the employees' column that reduces the number of employees by one. To implement the rule, the following WHERE clause must be defined:

```
old.employees - 1 = new.employees
```

Director_emp_track must be defined as the database procedure with the following parameters:

```
dname = old.director
```

This rule is fired by the manager_emp_track procedure, because it reduces the number of employees by one, but it is also fired if the manager table is updated directly.

The Copy Statement and Enforcing Rules

If you use the COPY statement on a table with rules defined, the table's rules are completely ignored. Table integrities are ignored in this same manner. How to effectively apply rules in this situation is described in The Copy Statement and Enforcing Integrities (see page 146).

Disable Rules

By default, rules are enabled. The SET NORULES statement enables you to turn off rules when necessary (for example, when using a utility that loads or unloads a database in which tables can be modified from scripts and files prior to their processing by applications).

To issue this statement, you must be the DBA of the database to which the session is connected.

The SET NORULES statement disables any rules that apply to statements executed during the session or to the tables affected by the statements. Existing rules as well as rules created during the session are disabled.

To re-enable rules, issue the SET RULES statement.

Warning! After you issue the SET NORULES statement, the DBMS does not enforce check and referential constraints on tables, nor does it enforce the check option for views.

For more information on using SET [NO]RULES, see the entry for the SET statement in the *SQL Reference Guide*.

Database Events

A *database event* enables an application or the DBMS to notify other applications that a specific event has occurred.

An *event* is any type of program-detectable occurrence.

Using database events, you can define an action that can be tied to a programmed response for the purpose of sequencing multiple actions or responding quickly to a specific database condition.

Working with Dbevent Objects

You can perform the following basic operations on dbevent (database event) objects:

- Create dbevent objects
- View existing dbevent objects, including the detailed properties of each individual object
- Drop dbevent objects

In SQL, you can accomplish these tasks using the CREATE DBEVENT and DROP DBEVENT statements. For details, see the *SQL Reference Guide*.

In VDBA, use the Dbevents branch for a particular database in the Database Object Manager window. For detailed steps, see the Procedures section of VDBA online help.

How Database Events Work

After a database event is defined for a table, it can be raised by all applications connected to the database, assuming appropriate privileges have been granted, as described in the *Security Guide*.

The event can be raised from interactive or embedded SQL applications, as a result of triggering a security alarm, or in a database procedure (where it can, in turn, be invoked by rules). It can also be received by all applications connected to the database and registered to receive the event.

In general, database events work as follows:

- An application or the DBMS raises an event, that is, issues a notification that a defined event has occurred.
- The DBMS notifies monitor applications that are registered to receive the event.
- The receiving application responds to the event by performing the action the monitor application designer specified when writing the program.

Note: You can also trace database events. For details, see the chapter “Using Monitoring and Tracing Tools” in the *System Administrator Guide*.

Raise an Event

To raise a database event, use the RAISE DBEVENT statement from interactive or embedded SQL applications or from within a database procedure.

A session can raise any event that is owned by the effective user, and any event for which the effective user, group, role, or public has been granted the raise privilege. For more information on granting privileges, see the *Security Guide*.

The RAISE DBEVENT statement requires you to specify an *event_name* parameter, which is the same as the value you enter in the Create Database Event dialog when you create the dbevent object using VDBA.

When the RAISE DBEVENT statement is issued, the DBMS sends an event message to all applications that are registered to receive the specified database event. If no applications are registered to receive the event, raising the event has no effect.

The optional *event_text* parameter is a string that can be used to pass context information or program handles to receiving applications. For example, use *event_text* to pass the name of the application that raised the event. You can retrieve this value using INQUIRE_SQL.

The WITH [NO]SHARE parameter enables you to specify which of the applications registered to receive the event are actually notified. If you specify WITH SHARE or omit this parameter, the DBMS notifies all registered applications when the event is raised. If you specify WITH NOSHARE, the DBMS notifies only the application that raised the event (assuming the program was also registered to receive the event).

If a transaction issues the RAISE DBEVENT statement and the transaction is subsequently rolled back, event queues are not affected by the rollback. The raised event remains queued to all sessions that registered for the event. The event queue is described in Receive an Event (see page 161).

For the complete statement syntax and additional information about using the RAISE DBEVENT statement, see the *SQL Reference Guide*.

Register to Receive an Event

To register to receive a database event, use the REGISTER DBEVENT statement from interactive or embedded SQL applications or from within a database procedure. For each event, the registration is in effect until the session removes the event registration or disconnects from the database.

A session can register for any event that is owned by the effective user, and any event for which the effective user, group, role, or public has been granted the register privilege. Sessions must register for each event to be received. For more information on granting privileges, see the *Security Guide*.

The DBMS issues an error if:

- A session attempts to register for a non-existent event.
- A session attempts to register for an event for which the session does not have register privilege.
- A session attempts to register twice for the same event.

If the REGISTER DBEVENT statement is issued from within a transaction that is subsequently rolled back, the registration is not rolled back.

For the complete statement syntax and additional information about using the REGISTER DBEVENT statement, see the *SQL Reference Guide*.

Receive an Event

To receive event information, an application must perform two steps:

1. Remove the next event from the session's event queue (using GET DBEVENT, or implicitly, using WHENEVER DBEVENT or SET_SQL DBEVENTHANDLER).
2. Inquire for event information (using INQUIRE_SQL).

Get the Next Event from the Event Queue

The GET DBEVENT statement gets the next event, if any, from the queue of events that have been raised and for which the application has registered.

For the complete statement syntax and additional information about using the GET DBEVENT, see the *SQL Reference Guide*.

Obtain Event Information

To obtain event information, your application must issue the INQUIRE_SQL statement. With this statement, you specify one or more parameters to determine the type of information to retrieve. For example, to retrieve the text specified in the *event_text* parameter when the event was raised, use INQUIRE_SQL (DBEVENTTEXT).

For the complete statement syntax and additional information about using the INQUIRE_SQL statement, see the *SQL Reference Guide*.

Example: Using Database Events with Rules

The following example illustrates the use of database events in conjunction with rules in a manufacturing application. In this case, an event is used to detect when a drill gets too hot; the drill is then taken offline:

1. Create a database event named drill_hot to be raised when the drill overheats.
2. Create a database procedure that raises the drill_hot event; the procedure is executed when the rule defined in step 3 is triggered.

   For example, the following procedure, take_drill_down, logs the time at which the drill was disabled and raises the drill_hot event:

   Parameters

   ```
   drill_id
   ```

   Statements

   ```
   insert into drill_log
     select date('now'), 'OFFLINE', drill.*
       from drill where id = :drill_id;
   raise dbevent drill_hot;
   ```

3. Create a rule named drill_hot that is triggered whenever the drill temperature is logged. (This presumes another application that monitors and logs drill temperatures. This is created in the next step.)

   For example, create a rule to execute the take_drill_down procedure (created in step 2) after any update operation in which the temperature column was changed. Using the following WHERE clause causes the rule to be fired if the temperature exceeded 500 degrees:

   ```
   new.temperature > 500
   ```

   The drill_id parameter must be passed as shown below:

   ```
   drill_id = drill.id
   ```

4. Finally, create an application that monitors the status of the drills.

   In the following example, the monitor application registers to receive the drill_hot event and checks for events. If the monitor application receives the drill_hot event, it sends mail to a supervisor and sends the signals required to disable the drill:

   ```
   exec sql register dbevent drill_hot;
   ...
   exec sql get dbevent
   exec sql inquire_sql (:evname = eventname, ....);
   if (evname = 'drill_hot') then
     send mail
     take drill offline
   endif;
   ```

The various pieces function together as follows:

1. The drill monitor application periodically logs the drill temperature to the drill log table.
2. When the drill monitor application logs a drill temperature in excess of 500 degrees, the drill_hot rule fires.
3. The drill_hot rule executes the take_drill_down database procedure, which raises the drill_hot event.
4. Finally, the event monitor process detects the drill_hot event, sends mail to notify the responsible user, and sends a signal that disables the overheated drill.

Remove an Event Registration

To remove a database event registration, use the REMOVE DBEVENT statement from interactive or embedded SQL applications or from within a database procedure.

Using REMOVE DBEVENT simply "unregisters" an application for a particular database event. The event is still defined for the database and can be received by other applications that are still registered.

After an event registration is removed, the DBMS does not notify the application when the specified event is raised. Pending event messages are not removed from the event queue.

For the complete statement syntax and additional information about using the REMOVE DBEVENT statement, see the *SQL Reference Guide*.

Drop Database Events

You can drop a dbevent object from the database, in which case it cannot be raised and applications cannot register to receive it. Pending event messages are not removed from the event queue.

If an event is dropped while applications are registered to receive it, the event registrations are not dropped until each application disconnects from the database or removes its registration for the dropped event. If the event is recreated (with the same name), it can again be received by registered applications.

Chapter 9: Choosing Storage Structures and Secondary Indexes

This section contains the following topics:

This chapter describes storage structures, secondary indexes, and keys. It will help you decide on the best structure and corresponding options to suit your needs.

Storage Structure Terminology

A *storage structure* is a file arrangement providing a way to access data in a database table.

Keyed storage structures provide fast access to a particular row or set of rows in a database table.

A *key* is the field or fields that the table is indexed on. Specifying this key gives you quick access to the rows you are looking for.

An *index* contains the contents of the key fields.

A *secondary index* allows you to specify an additional key.

Storage Structure and Performance

Ingres provides multiple types of storage structures. Each storage structure provides optimal performance for particular types of queries and applications. Choosing the best storage structure is essential to maintaining good performance.

When you create or modify a table, you can choose the appropriate storage structure and specify options to fine-tune the structure.

Types of Storage Structures

The types of storage structures are summarized here:

Heap

The non-keyed storage structure with sequential data entry and access. There is also a compressed heap structure (cheap) with trailing blanks removed.

Hash

A keyed storage structure with algorithmically chosen addresses based on key data values. There is also a compressed hash structure (chash) with trailing blanks removed.

ISAM

A keyed storage structure in which data is sorted by values in key columns for fast access. The index is static and needs remodification as the table grows. There is also a compressed ISAM structure (cISAM) with trailing blanks removed.

B-tree

A keyed storage structure in which data is sorted by values in key columns, but the index is dynamic and grows as the table grows. There is also a compressed B-tree structure (cB-tree) with trailing blanks removed.

For more information on the compressed structure for each of the above types, see the chapter "Maintaining Storage Structures."

Another storage structure, R-tree, can be used only on secondary indexes, as described in R-tree Secondary Index (see page 197).

Default Storage Structure of New Tables

The default storage structure of a newly created base table is determined by the setting of the configuration parameter table_auto_structure in combination with the presence of constraint definitions in the CREATE TABLE statement.

When table_auto_structure is ON, the storage structure of a base table is automatically determined based on the syntax used for the CREATE TABLE statement. If the CREATE TABLE statement includes at least a primary key, unique constraint, or referential (foreign key) constraint, the base table structure is set to B-tree on the constrained columns and the usual secondary index is not built for the constraint.

If the table definition includes more than one constraint, it chooses the primary key constraint over a unique constraint, and the first unique constraint over any referential constraint. For primary key or unique constraints, it also adds the UNIQUE_SCOPE=STATEMENT attribute to the base table structure. A dependency is added between the constraint and the base table structure so that the constraint must be explicitly dropped and re-added if the base table structure is modified.

When table_auto_structure is OFF or if there are no accompanying constraint definitions, the default storage structure of all new base tables is heap.

Heap Storage Structure

In a heap structure, the table has no key—it is simply a heap of data. When you add a row, it is added to the end of the heap. This makes heap the fastest storage structure to use when you are initially loading tables or adding a large quantity of data.

However, when you want to retrieve a particular row from a heap table, you must search through every row in the table looking for rows that qualify. This makes heap relatively slow for retrieval if tables have more than a few pages. For more information, see the chapter “Maintaining Storage Structures.”

Note: The heapsort structure is like heap, but with the rows sorted and duplicates removed (unless duplicates are allowed).

Structure of a Heap Table

A heap table consists of a chain of pages. The layout of the sample heap table, employee, is shown below:

```
       empno  name        age  salary     comment
       +--------------------------------------------
Page 0 |   17| Shigio     | 29| 28000.000|
       |    9| Blumberg   | 33| 32000.000|
       |   26| Stover     | 38| 35000.000|
       |    1| Mandic     | 46| 43000.000|
       |--------------------------------------------
Page 1 |   18| Giller     | 47| 46000.000|
       |   10| Ming       | 23| 22000.000|
       |   27| Curry      | 34| 32000.000|
       |    2| Ross       | 50| 55000.000|
       |--------------------------------------------
Page 2 |   19| McTigue    | 44| 41000.000|
       |   11| Robinson   | 64| 80000.000|
       |   28| Kay        | 41| 38000.000|
       |    3| Stein      | 44| 40000.000|
       |--------------------------------------------
Page 3 |   20| Cameron    | 37| 35000.000|
       |   12| Saxena     | 24| 22000.000|
       |   29| Ramos      | 31| 30000.000|
       |    4| Stannich   | 36| 33000.000|
       |--------------------------------------------
Page 4 |   21| Huber      | 35| 32000.000|
       |   13| Clark      | 43| 40000.000|
       |   30| Brodie     | 42| 40000.000|
       |    5| Verducci   | 55| 55000.000|
       |--------------------------------------------
Page 5 |   22| Zimmerman  | 26| 25000.000|
       |   14| Kreseski   | 25| 24000.000|
       |   31| Smith      | 20| 10000.000|
       |    6| Aitken     | 49| 50000.000|
       |--------------------------------------------
Page 6 |   23| Gordon     | 28| 27000.000|
       |   15| Green      | 27| 26000.000|
       |    7| Curan      | 30| 30000.000|   Fire
       |   24| Sabel      | 21| 21000.000|
       |--------------------------------------------
Page 7 |   16| Gregori    | 32| 31000.000|
       |    8| McShane    | 22| 22000.000|
       |   25| Sullivan   | 38| 35000.000|
       |
       +--------------------------------------------
```

Because table scans are expensive, heap is not a good structure to use while querying large tables. A retrieval of this type must look at every page in the employee table:

```
Select * from employee
   where employee.name = 'Sullivan';
```

A retrieval like this also scans the entire table, even though Shigio's record is the first row of the first page:

```
Select * from employee
   where employee.name = 'Shigio';
```

Because heap tables do not eliminate duplicate rows, the entire table must be scanned in case there is another employee named Shigio on another page in the table.

Heap as Structure for Loading Data

If the configuration parameter auto_table_structure is set to OFF, heap is used as the default storage structure when a table is first created, because it is assumed that a newly created table is likely to be loaded with data.

Loading is optimized by not doing "per row" logging. Therefore, you must load into an empty table. This can be a table that was just created and into which no data has ever been added or deleted. Or it can be an existing table that was truncated by clicking Delete All Data in the Modify Table Structure dialog or by using the MODIFY TO TRUNCATE statement.

The empty table must also have the following characteristics:

- The table must not be journaled or have secondary indexes.
- The table must not have system-maintained keys.
- You must have an exclusive lock on the table.

Heap is also the best structure to use for adding data. Additions to a heap table progress quickly because inserted rows are added to the end of the heap. There is no overhead of calculating what page the row is on. The disadvantage is that the heap structure does not make use of deleted row space except at the end of the table.

Aside from compressed storage structures, the heap structure produces tables with the smallest number of pages. This is because every page in a heap table is filled as completely as possible. This is referred to as a 100% fill factor. A heap table is approximately half the size of the same table modified to hash because hash uses a 50% default fill factor instead of 100%.

After loading or adding the data, you can modify the table to another storage structure. (Do not modify an empty table to another storage structure before loading the data.)

To free deleted space, remodify the table to heap using the Modify Table Structure dialog or the modify statement.

Very small tables can usually be left as heap tables. If the table fits on one to five pages as a heap, there is no speed advantage to modifying it to a different structure.

Note: The heap structure is sometimes used for large tables in conjunction with a secondary index. This can be useful in a situation where the table is so large it cannot be modified, but an accelerated access method is needed.

When to Use Heap

Heap is a good storage structure to use in any of these cases:

- You are bulk-loading data into the table.
- The table is only a few pages long (a lookup table).
- You always retrieve every row in the table without sorting.
- You are using a secondary index on a large table and must conserve space.

Do not use heap for large tables when query performance is the top priority. Heap is also a poor storage structure to use if you look up particular rows by key value.

Heap Troubleshooting

The following are problems encountered with heap storage structure, and their solutions:

Problem	**Solution**
Access is slow on a table created from another table (for example, using the CREATE TABLE...AS SELECT statement or the Create Table as Select check box in the Create Table dialog).	Change the storage structure of the table from which you are selecting the data, or specify a storage structure other than heap for the table you are creating.
Space once used by deleted rows is never reused.	Modify the table to reclaim the deleted row space (for example, using the modify statement or the Modify Table

Problem	Solution
	Structure dialog). In this case, you can still choose heap as the storage structure.
Selects and updates are slow.	If the table is not small, modify it to another storage structure. Heap is used only for small tables because the entire table is always scanned. Alternatively, you can create a secondary index.
Inserts are not concurrent.	Use row locking if the page size is greater than 4 KB, or modify to another structure. All inserts to a heap table are sent to the last page.

Hash Storage Structure

Hash is the keyed storage structure that calculates a placement number or address by applying a *hashing algorithm* to the key data value. A hashing algorithm is a function that does mathematical computations to a piece of data to produce a number. It always produces the same number for the same piece of data.

Hash is the fastest access method for exact match queries (that is, with no pattern matching). A quick calculation is used to determine which pages to search, but there is no additional I/O necessary for index scanning, as there is in an ISAM or B-tree table. However, hash is more limited in the types of queries it can handle, because the hashing algorithm is not useful in looking for ranges of values, handling partial key restrictions, or doing pattern matching. For these types of queries, the entire table must be scanned.

Using the Modify Table Structure dialog or the MODIFY statement, you can change any table to the hash storage structure. When you modify a table to hash, you should specify a key; otherwise, the first column is used as a key.

Modifying a table to hash involves several calculations. Taking the number of rows currently in the table, and calculating how many rows can fit on a 2000-byte page, modify calculates how many *main pages* are necessary. (Main pages are data pages where the rows are actually stored.)

To help the hashing algorithm distribute the data evenly, as well as to allow plenty of room to add new data, this figure is doubled (referred to as 50% fill factor). This is the number of main pages assigned to the table. The hashing algorithm decides on which main page the row resides by calculating its hashing address.

Structure of a Hash Table

An example illustrates how a hash table is structured and what *hashing* means:

- The example uses an employee table that has 31 rows, 500 bytes each.
- The table is modified to hash on the age field, using the Structure of Table dialog. For a full description of modify procedures, see the chapter “Maintaining Storage Structures.” You can also use this MODIFY statement:

  ```
  modify employee to hash on age;
  ```

The number of main pages needed is calculated. The number chosen is always at least seven, no matter how small the table is. The number of main pages chosen is approximately twice the number of pages required if the table were a heap. Normally hash uses a 50% fill factor, although if the row width is greater than 1000 bytes, it uses a 100% fill factor.

The calculation used is this:

```
Main pages =  (Rows_in_Table / Rows_per_page) * 2

31 rows_in_table / 4 rows_per_page = 8 (round up)
   8 * 2 = 16;

Main pages for employee table = 16
```

The main pages calculation is checked against the Min Pages and Max Pages values. If these were specified, the result must fall in this range.

When a table is modified to hash, a skeletal table is set up with an appropriate number of main pages. Although 16 pages can actually be used, as shown in the calculation above, for illustration purposes assume 10 main pages are chosen. The table is built by placing each row on the page where its key hashes.

The chart in the following example illustrates what a table looks like after modifying to hash on age. Remember that the actual hashing function is a complex algorithm that supports all of the data types. For simplicity, however, the following examples use the module function as a hashing algorithm.

Here is an example of a hashing function:

```
Main Page = Key MOD Main_Pages

Ross,     Age 50     50 mod 10 = 0; hashes to page 0
McShane,  Age 22     22 mod 10= 2; hashes to page 2
.
.
.
```

After this hashing process is completed for all 31 rows, the table looks like this:

```
        +----------------------------+
Page 0  |     50|Ross      | 55000.000|
        |     20|Smith     | 10000.000|
        |     30|Curan     | 30000.000|
        |     20|Sabel     | 21000.000|
        |----------------------------|
Page 1  |                            |
        |                            |
        |                            |
        |                            |
        |----------------------------|
Page 2  |     22|McShane   | 22000.000|
        |     32|Gregori   | 31000.000|
        |     42|Brodie    | 40000.000|
        |                            |
        |----------------------------|   Overflow Chain for Page 3
Page 3  |     33|Blumberg  | 32000.000|    |----------------------------|
        |     43|Clark     | 40000.000|-->|     23|Ramos     | 30000.000|
        |     23|Ming      | 22000.000|    |     53|McTigue   | 41000.000|
        |     43|Kay       | 38000.000|    |----------------------------|
        |----------------------------|
Page 4  |     24|Saxena    | 22000.000|
        |     34|Curry     | 32000.000|
        |     44|Stein     | 40000.000|
        |     64|Robinson  | 80000.000|
        |----------------------------|
Page 5  |     55|Verducci  | 55000.000|
        |     35|Huber     | 32000.000|
        |     25|Kreseski  | 24000.000|
        |                            |
        |----------------------------|
Page 6  |     26|Zimmerman | 25000.000|
        |     46|Mandic    | 43000.000|
        |     36|Stannich  | 33000.000|
        |                            |
        |----------------------------|
Page 7  |     37|Cameron   | 35000.000|
        |     47|Giller    | 46000.000|
        |     27|Green     | 26000.000|
        |                            |
        |----------------------------|
Page 8  |     38|Stover    | 35000.000|
        |     38|Sullivan  | 35000.000|
        |     28|Gordon    | 27000.000|
        |                            |
        |----------------------------|
Page 9  |     49|Aitken    | 50000.000|
        |     29|Shigio    | 28000.000|
        |                            |
        |                            |
        +----------------------------+
```

To retrieve the employee data about employees who are 49 years old:

```
Select * from employee
  Where employee.age = 49;
```

The table is hashed on age, and the qualification has specified an age. The hashing algorithm is used to determine the main page on which rows with ages of 49 are located:

```
49 mod 10 = 9
```

The lookup goes directly to Page 9, instead of looking through the entire table, and returns the row requested.

To find all employees who are age 53, the calculation is:

```
53 mod 10 = 3
```

These rows are found on main Page 3. However, a search through the page, looking for 53, shows it is not there. There is an overflow page, though, and searching this page finds the row. Overflow pages are associated with a particular main page. They can slow down processing time because searches are required not only on the main page but the overflow chain connected to the main page as well.

Inserts, updates, and deletes work the same way retrievals do. If you want to append a new row, the row is placed on the page the new employee's age hashes to. Therefore, if you add an employee with age 22, 22 mod 10 is 2, so this employee is placed on main Page 2.

To find all employees who are older than 50, there is no way of directly locating these rows using the hashing algorithm; you must hash on every possible value greater than 50. Instead, the table is treated as a heap table and every page is scanned, starting from Page 0 through to Page 9 including all overflow pages, looking for qualifying rows.

To retrieve the row where the employee's name is Shigio, does the age key help? Because the table is not hashed on name, and Shigio's age is unknown, the entire table must be scanned, from Page 0 through Page 9, looking for rows where the employee name is Shigio. This retrieval treats the table like a heap table, scanning every main and overflow page. To retrieve a row you need without scanning the entire table, specify the value of the key for the row.

Retrievals Supported by Hash

The hash storage structure allows multi-column keys, but every column in the key must be specified in a query to take advantage of the hash access method. For instance, to hash the employee table on both age and name, use the Structure of Table dialog.

Alternatively, use the following MODIFY statement:

```
modify employee to hash on age,name;
```

The following queries make use of the hash key:

```
select * from employee
  where employee.age = 28
  and employee.name = 'Gordon';

select * from employee
  where employee.age = 28
    and employee.name = 'Gordon'
  or employee.age = 29
    and employee.name = 'Quan';
```

The next queries do not use the hash key, because the entire key has not been specified:

```
select * from employee
  where employee.age = 28;

select * from employee
  where employee.name = 'Gordon';

select * from employee
  where employee.age = 28
  and employee.name like 'Gor%';

select * from employee
  where employee.age = 28
  or employee.name = 'Gordon';
```

When to Use Hash

Hash is the fastest structure to use when you specify an exact match of the whole key value. Hash does not efficiently support pattern matching, range searches, or partial key specification with multi-column keys. For these queries the entire table must be scanned.

Hash is a good storage structure to use if you always retrieve the rows based on a known key value, such as order number or employee number.

Hash is a poor storage structure to use in any of these cases:

- You use pattern matching.
- You retrieve ranges of values.
- You specify part of a multi-column key.

Hash Troubleshooting

The following are problems encountered with hash storage structure, and their solutions:

Problem	**Solution**
Pattern matching and range scans used; performance slow.	Use ISAM or B-tree instead.
Partial key of multi-column key used; performance slow.	Use ISAM or B-tree instead.
Overflow pages occur in table after adding rows.	Remodify.
Overflow pages occur in newly modified table.	If key is repetitive, this is normal but undesirable. If key is unique, hashing algorithm does not distribute data well; try increasing minpages. If column is a character column that only partially varies (for example, AAAAA1 AAAAA2), consider using ISAM instead.

ISAM Storage Structure

ISAM is a keyed storage structure in which data is sorted by the value in the key column, and the index is static.

ISAM is a more versatile storage structure than hash. It supports pattern matching, range scans, and partial key specification, as well as exact match retrievals.

ISAM tables use a static index that points to a static number of main pages. The index contains key ranges and pointers either to other index pages or to the data page where rows with that key range are found.

Using the Modify Table Structure dialog or the MODIFY statement, you can change any table to the ISAM storage structure. When you modify a table to ISAM, you must specify a key; otherwise, the first column is used as a key.

Structure of an ISAM Table

Here is a simple example that illustrates how the ISAM structure works. The employee table, which has 31 rows with a byte-width of 500, is modified to ISAM on employee number. The results are shown in the following table:

```
                      empno name      age    salary
Index Pages          +-----------------------------------------------
                     |1   |Mandic     |46|   43000.000|
            <=4      |2   |Ross       |50|   55000.000|Data Page 1
            =Page 1  |3   |Stein      |44|   40000.000|
     <=4             |4   |Stannich   |36|   33000.000|
     ?               |-----------------------------------------------
     >=5    >4 and   |5   |Verducci   |55|   55000.000|
            <=8      |6   |Aitken     |49|   50000.000|Data Page 2
<=8         =Page 2  |7   |Curan      |30|   30000.000|
?                    |8   |McShane    |22|   22000.000|
>8                   |-----------------------------------------------
            >8 and   |9   |Blumberg   |33|   32000.000|
            <=12     |10  |Ming       |23|   22000.000|Data Page 3
            =Page 3  |11  |Robinson   |64|   80000.000|
     <=12            |12  |Saxena     |24|   22000.000|
     ?               |-----------------------------------------------
     >=13   >12 and  |13  |Clark      |43|   40000.000|
            <=16     |14  |Kreseski   |25|   24000.000|Data Page 4
<=16        =Page 4  |15  |Green      |27|   26000.000|
?                    |16  |Gregori    |32|   31000.000|
>16                  |-----------------------------------------------
                     |17  |Shigio     |35|   32000.000|
     <=20   >16 and  |18  |Giller     |47|   46000.000|Data Page 5
     ?      <=20     |19  |McTigue    |44|   41000.000|
     >20    =Page 5  |20  |Cameron    |37|   35000.000|
                     |-----------------------------------------------
                     |21  |Huber      |35|   32000.000|
<=24        >20 and  |22  |Zimmerman  |26|   25000.000|Data Page 6
?           <=24     |23  |Gordon     |28|   27000.000|
>24         =Page 6  |24  |Sabel      |21|   21000.000|
                     |-----------------------------------------------
                     |25  |Sullivan   |38|   35000.000|
            >24and   |26  |Stover     |38|   35000.000|Data Page7
     <=28   <=28     |27  |Curry      |34|   32000.000|
     ?      =Page7   |28  |Kay        |41|   38000.000|
     >28             |-----------------------------------------------
                     |29  |Ramos      |31|   30000.000|
            >28      |30  |Brodie     |42|   40000.000|Data Page8
            =Page8   |31  |Smith      |20|   10000.000|
                     |
                     +-----------------------------------------------
```

Suppose you want to retrieve the employee data about employee number 11. Starting at the beginning of the index (at the left in the example), follow the index over to data Page 3, which contains rows of employees with employee numbers greater than 8 and less than or equal to 12. Scanning this page, you find employee number 11’s row.

If you want to find all employees with employee numbers greater than 24, use the index, which directs you to Page 7, where you begin scanning the remainder of the table looking for qualifying rows.

To retrieve the row where the employee's name is Shigio, empno key does not help, because the index was constructed on empno and not on name. You must scan the entire table, from Page 0 through Page 9.

To append a new employee with an empno of 32, the search scans through the index to the largest key value less than 32. On the page with that key (Page 8), the new row is placed on the first available space on that page. If no room is left on that page, the row is placed on an overflow page.

Retrievals Supported by ISAM

ISAM can limit a scan if you specify at least the leftmost part of the key for the desired rows. ISAM also limits the pages scanned if you are looking for ranges of the key.

- If the key is a character key, ISAM supports character matching with limited scan if you specify at least the leftmost part of the key.
- If the key is a multi-column key, ISAM limits the pages scanned only if you specify at least the leftmost part of the key.

For instance, assume you modified the employee table to ISAM on name and age using the Structure of Table dialog. Alternatively, you can use the following MODIFY statement:

```
modify employee to ISAM on name, age;
```

The following retrievals make use of the ISAM key:

```
select * from employee
  where employee.name like 'S%';

select * from employee
   where employee.name = 'Shigio'
    and employee.age > 30;
```

In contrast, the following retrievals do not make use of the ISAM key, because the leftmost part of the key (name) is not restricted:

```
select * from employee
  where employee.age = 32;

select * from employee
  where employee.name like '%S'
  and employee.age = 32;

select * from employee
  where employee.name like '%higio%';
```

When to Use ISAM

ISAM is a versatile storage structure because it supports both exact match and range retrievals. ISAM indexes and main pages are static—if you are appending many rows, remodify to avoid overflow pages. For tables that are mostly static, ISAM can be preferable to B-tree.

Because ISAM indexes are static, no locking needs to be done on the ISAM index. In a heavily concurrent update environment, this feature makes ISAM more appealing than B-tree, where pages of the index must be locked when splitting or updating occurs.

ISAM is a good storage structure to use when the table is relatively static, and retrievals tend to use any of the following:

- Pattern matching
- Ranges of key values
- Only the leftmost part of a multi-column key

ISAM is a poor storage structure to use in any of these cases, which causes overflow pages:

- The table is growing at a rapid rate.
- The table is too large to modify.
- The key is sequential, that is, each key number is higher than the last and the data is not static. This is because adding data with sequential keys adds a lot of overflow pages at the last main page.

ISAM Troubleshooting

The following are problems encountered with the ISAM storage structure, and their solutions:

Problem	Solution
You try to use pattern matching, but do not specify the leftmost character.	*F* does not use the ISAM index, whereas F* does. If you cannot modify the search condition, the entire table must be scanned.
You try to use just part of a multi-column key, but do not specify the leftmost column.	If you cannot modify the search condition, create a secondary index with only the columns on which you are searching.
The table is growing quickly and new rows are added to overflow pages.	Use B-tree instead.

B-tree Storage Structure

B-tree is the keyed storage structure in which data is sorted by value in the key column for fast access on the exact value and range retrievals, and the index is dynamic. It is the most versatile storage structure. The B-tree structure allows for keyed access and supports range searches and pattern matching. The B-tree index is dynamic, growing as the table grows. This eliminates the overflow problems that static structures like ISAM and hash present as they grow. It is possible for a B-tree table using 2K pages to develop overflow from a leaf page with sufficient duplicate key values. B-tree secondary indexes never develop overflow because the key is always physically unique (it includes the tidp column in non-logically-unique indexes). B-tree also allows for maximum concurrent use of the table.

B-tree design incorporates a sparse index that points to pages in a leaf level. The leaf level is a dense index that points to the rows on the data pages in the table. The benefit of this indexing approach is that it minimizes splitting cost: when splitting does occur, the actual data rows need not move. Only the leaf and index levels require reorganization, as described in Index Growth in a B-tree Table (see page 185).

If the configuration parameter table_auto_structure is set to ON, and if the CREATE TABLE statement includes at least a primary key, unique constraint, or referential (foreign key) constraint, the base table structure is set to B-tree on the constrained columns and the secondary index is not built.

Structure of a B-tree Table

A B-tree can be viewed as four separate parts:

- A free list header page, which is used to keep track of allocated pages that are not currently being used
- One or more index pages, which contain leaf page numbers and the range of key values to expect on each leaf page
- One or more leaf pages, which identify the data page and row where the data is stored
- One or more data pages, where the user data is actually stored

The smallest B-tree has four pages, one of each type.

Note: If a secondary index is modified to B-tree, it cannot contain data pages. Instead, the leaf pages of the secondary index reference the main table's data pages. For more information, see Secondary Indexes (see page 194).

The index level is similar to the ISAM index, except that the ISAM index points to data pages, whereas the B-tree index level points to leaf pages. The number of index pages is dependent on the width of the key and the number of leaf pages, because eventually the index pages point to a particular leaf page. Usually the index level is small, because it needs to point to only the leaf pages.

The leaf page level is considered a dense index because it tells the location of every row in the table. In dense indexes, rows on data pages do not move during a split; that causes their tids to change. Tids identify every row on every data page. For a complete discussion of tids, see Tids (see page 202).

The index level is considered a sparse index, because it contains only a key value and a pointer to a page.

The following diagram illustrates the three B-tree levels: index page, leaf page, and data page. It illustrates the relationship between the three levels, but cannot be realistic. In actuality, if the key width name were only 30 characters, the row width were 500 bytes, and there were only 31 employees, this B-tree has only a free list header page, one index page, one leaf page, and 8 data pages (instead of 4 leaf pages and 3 index pages).

```
                                       +----------------------------------+
                                       |              ROOT                |
                                       |                                  |
                                       |          <= McShane              |
INDEX PAGE                             +----------------------------------+
LEVEL                                       /                    \
                                           /                      \
+-------------------------------------+ +---------------------------------+
| Key                        Leaf Page | | Key                  Leaf Page   |
|                                     | |                                 |
| <= Giller                         1   | | > McShane <= Shigio      3      |
| > Giller  <= McShane              2   | | > Shigio                 4      |
+-------------------------------------+ +---------------------------------+

LEAF PAGE LEVEL
Leaf Page   1        Leaf Page    2        Leaf Page    3     Leaf Page   4
Aitken    1,0        Gordon     3,0        McTigue    5,0     Smith     7,0
Blumberg  1,1        Green      3,3        Ming       5,1     Stannich  7,1
Brodie    1,3        Gregori    3,2        Ramos      5,2     Stein     7,2
Cameron   1,2        Huber      3,1        Robinson   5,3     Stover    7,3
Clark     2,0        Kay        4,0        Ross       6,0     Sullivan  8,0
Curan     2,1        Kreseski   4,1        Sabel      6,1     Verducci  8,1
Curry     2,2        Mandic     4,2        Saxena     6,2     Zimmerman 8,2
Giller    2,3        McShane    4,3        Shigio     6,3

DATA PAGE LEVEL
              +-------------------------------------------+
Page 1    0   |Aitken    |         1|     49| 50000.000   |
          1   |Blumberg  |         2|     33| 32000.000   |
          2   |Cameron   |         4|     37| 35000.000   |
          3   |Brodie    |         3|     42| 40000.000   |
              |-------------------------------------------|
Page 2    0   |Clark     |         5|     43| 40000.000   | Associated Data
          1   |Curan     |         6|     30| 30000.000   | Page for Leaf
          2   |Curry     |         7|     34| 32000.000   | Page 1
          3   |Giller    |         8|     47| 46000.000   |
              |-------------------------------------------|
Page 3    0   |Gordon    |         9|     28| 27000.000   |
          1   |Huber     |        12|     35| 32000.000   |
          2   |Gregori   |        11|     32| 31000.000   |
          3   |Green     |        10|     27| 26000.000   |
              |-------------------------------------------|
Page 4    0   |Kay       |        13|     41| 38000.000   | Associated Data
          1   |Kreseski  |        14|     25| 24000.000   | Page for Leaf
          2   |Mandic    |        15|     46| 43000.000   | Page 2
          3   |McShane   |        16|     22| 22000.000   |
              |-------------------------------------------|
Page 5    0   +McTigue   |        17|     44| 41000.000   +
```

To look for an employee named Kay, the search starts from the root node, where a name that precedes McShane in the alphabet directs you down the left side of the index.

The index page on the left shows that leaf Page 2 is the appropriate page on which to look, because Kay comes between Giller and McShane in the alphabet.

On leaf Page 2, Kay's record is identified as being on data Page 4, row 0. Going directly to data Page 4, row 0, Kay's record is located.

Associated Data Pages in a B-tree Table

Every leaf page has an *associated data page*. The associated data page is where new rows are added. A leaf page can actually point to several different pages, but new data is only added to the associated data page. When an associated data page fills up, a new associated data page is attached to the leaf page. If you delete rows that exist on the current associated data page, the deleted space is reused.

Having one associated data page per leaf page provides a good chance for rows with similar key ranges to exist on the same data page, thereby increasing the likelihood that data references occur on the same data page.

Index Growth in a B-tree Table

The major difference between ISAM and B-tree is that the B-tree index grows as the table grows. If you added these five new employees to the ISAM employee table, keyed on name: Zanadu, Zentura, Zilla, Zorro, Zumu, these names are put on the last page of the ISAM table. Because they do not all fit on the last page, they are put onto an overflow page attached to the last page.

If you added these five new employees to the B-tree table, you add the new names to the appropriate leaf page (Page 4, in this case) and their records go on the associated data page for leaf Page 4. Because the associated data page fills up, a new associated data page is assigned to Page 4. If the leaf page is full, and cannot hold all five names, the leaf page splits into two leaf pages, and a reference to the new leaf page in the index is added. If the index page can no longer hold a reference to another leaf page, the index is split as well.

Splitting in a B-tree Table

Splitting occurs fairly frequently while the table is small and growing. As the table gets larger, splitting occurs less frequently (unless a sequential key is used) and usually only in the leaf or lowest index level.

Repeated inserts into the right-most leaf of a B-tree table create empty leaf pages rather than half-full ones. This improves insert and retrieval performance, and increases disk space efficiency.

Locking and B-tree Tables

During normal B-tree traversal, leaf and data pages are logically locked until the end of the transaction. B-tree index pages are only temporarily locked during query execution. The index page lock is released after the page has been searched.

When searching the B-tree index, ladder locking is used: a lock is taken on the first index page, which points to another index page. The next index page is locked and, once it is locked, the first lock is dropped, and so on down the index to the leaf level.

The locking system always locks the leaf and data pages when accessing B-tree tables. Because of this, locking in a B-tree table requires twice as many locks as locking an ISAM or hash table. It is wise to set the maxlocks escalation factor higher than the default when using the B-tree storage structure. For details, see the SET LOCKMODE statement in the *SQL Reference Guide*.

Sorted Order in a B-tree Table

In the diagram in Structure of a B-tree Table (see page 183), rows for Huber and Green are not in sorted order on the data page. This happens if Huber's record was appended before Green's. They both end up on the same data page, but slightly out of order. This happens in ISAM as well. However, if you tried the following retrieval, you retrieve the rows in sorted order if the employee table was a B-tree. This is because the leaf pages are used to point to the data rows, and the leaf pages maintain a sorted sequence:

```
select * from employee
  where employee.name like 'G%';
```

The data on the data pages is not guaranteed to be sorted, but the access, which is always through the leaf pages, guarantees that the data is retrieved in sorted order. (This is not true for ISAM.)

Because the leaf entries are in sorted order, the maximum aggregate for a B-tree key does not require a table scan. Instead the index is read backwards.

Deleted Rows in a B-tree Table

If rows are deleted on the associated data page, the space is reused the next time a row is appended to that page. If rows are deleted from a data page that is no longer associated, the space is not reused. If all the rows on a non-associated data page are deleted, the page is immediately added to the free list and becomes available for reuse.

Note: The only way to free up unused data pages completely and return disk space to the operating system is to change the storage structure to B-tree. You can do this using the Modify Table Structure dialog or using the MODIFY statement.

The reason that deleted space on a non-associated data page is not automatically reused is to speed the append operation. Appending to one particular page (the "associated data page") is faster than tracking and checking all the available free space on non-associated data pages; appending to the associated data page also provides better key clustering when data addition occurs in sorted key order. Because appends generally occur more frequently than deletes, preserving the performance of the append operation seems wiser than reusing deleted space from non-associated data pages.

When to Use B-tree

B-tree is the most versatile storage structure, as it supports both exact match and range retrievals and includes a dynamic index, so that frequent remodification is not necessary.

B-tree is a good storage structure to use in any of these cases:

- The table is growing at a rapid rate.
- You use pattern matching.
- You retrieve ranges of key values.
- You retrieve using only the leftmost part of a multi-column key.

B-tree is a poor storage structure to use if:

- The table is relatively static.
- The table is small, static, and access is heavily concurrent.

B-tree Troubleshooting

The following are problems encountered with the B-tree storage structure, and their solutions:

Problem	Solution
You tried to use pattern matching, but did not specify the leftmost character.	Specify the leftmost part of the key; *F* does not use the B-tree index, but F* does. If you cannot modify the search condition, the entire table must be scanned.
You tried to use just part of a multi-column key, but did not specify the leftmost column.	Specify the leftmost column of the multi-column key. If you cannot modify the search condition, create a secondary index with only the columns on which you are searching.
You are deleting frequently, as well as adding data.	To reclaim space, periodically select Shrink B-tree Index in the Modify Table Structure dialog, or use the MODIFY TO MERGE or MODIFY statements.

ISAM or B-tree?

The B-tree and ISAM data structures share many of the same advantages over the other storage structures, but they differ in important respects.

When to Choose ISAM over B-tree

The ISAM storage structure has the following advantages over B-tree:

- ISAM is better for static tables (ones that have no updates on key fields, appends, or deletes) where no overflow chains exist.
- ISAM requires fewer disk operations to visit a data page than B-tree, because B-tree has an additional leaf level.
- ISAM is much better for small tables. B-tree requires a minimum of a free list header page, a root page, a leaf page, and a data page. ISAM requires only a root and a data page. B-trees for less than 10 to 15 pages are better stored as ISAM. B-tree tables take up more space than do ISAM tables; this is most noticeable when tables are small.
- ISAM requires no locking in the index pages, while B-tree incurs index locking; therefore concurrent performance in the index of a B-tree is not as good as concurrent performance in the index pages of an ISAM. However, concurrent usage in B-tree data pages is better than concurrent usage in ISAM data pages if the ISAM table has long overflow chains.

When to Choose B-tree over ISAM

The B-tree storage structure has the following advantages over ISAM:

- B-tree is essential in tables that are growing at a rate that quickly causes overflow in an ISAM structure (for example, situations where there are ever-increasing keys).
- B-tree is better when sorting on the key is required, because sequential access (for example, SELECT * FROM emp) to data in B-tree is automatic; there is no need to add a SORT clause to queries if you are sorting on the primary key. B-tree also eliminates sorting of the joining column when joining on key columns; sort-merge queries are more efficient if the tables joined are B-tree.

Storage Structure Comparison Summary

The following chart is a quick reference for deciding which storage structure to use.

Why a particular storage structure is good or bad for the condition listed is described under the section Storage Structures and Performance (see page 205). Information on secondary indexes is described in Secondary Indexes (see page 194).

Ratings in the following chart are as follows: 1-Excellent, 2-Good, 3-OK, 4-Bad, N/A-not applicable.

Requirement	**Heap**	**Hash**	**ISAM**	**B-tree**
Pattern matching	4	4	1	1
Range searches	4	4	1	1
Exact-match keyed retrievals	4	1	2	2
Sorted data (without sort-by)	4	4	2	1
Concurrent updates	4	1	1	2
Addition of data without needing to modify	2	3	3	1
Sequential addition of data (incremental key)	1*	2	4	1
Initial bulk copying of data	1	2	2	2
Table growth: none, static	N/A	1	1	2
Table growth: some, periodically plan to modify	N/A	1	1	2
Table growth: great deal — too fast to modify	3	3	3	1
Table size: small (under 15 main pages)	2	1	1	3
Table size: medium (disk space available for any modify)	4	1	1	1
Frequent deletions	4	1	1	3
Frequent updates	4	1	1	2
Secondary index structure	N/A	1	1	1

* Refers to secondary indexes used with a heap table.

Keys

Structures that provide fast access to particular rows or sets of rows require that one or more columns be specified as the *key* of the table. The *key column* or columns are used to index the table. When specifying a value for this key, a partial value (the leftmost part of the key) is allowed unless the structure is hash.

Key Columns

When a key value is specified, instead of scanning the entire table, the search uses the index (or *hashes the key*) to go directly to the page in the table where the row with that key resides.

Choosing which columns to use as key columns is not always clear cut. To understand what a key column does, let us look again at the employee table. Consider the query:

```
select * from employee
  where name = 'Shigio';
```

The column called name (assuming it is unique) is a good candidate for the key for the employee table. If the employee table is keyed on name, finding the employee record where the name is Shigio is faster than scanning the entire table.

Good columns for keys are columns referenced in the WHERE clause portion of the query, not the target list. Columns that restrict the number of rows returned and joining columns, demonstrated in the two examples below, are candidates for keys:

```
where name = 'Shigio'
where e.dept = d.dept
```

A join qualification by itself is not restrictive, so if there also exists a restrictive qualification in the WHERE clause, choose the restriction as the key column. For example:

```
select empno from employee
  where employee.name = dept.manager
  and dept.name = 'Technical Support';
```

The most restrictive qualification in this WHERE clause is:

```
dept.name = 'Technical Support'
```

The dept table is keyed on name. Keying dept on manager is not necessary for this query, because once the row for the department named Technical Support is identified, you know the manager. The employee table is also keyed on name, because once the manager of the dept table is known, the search can do a keyed lookup into the employee table. Empno is not a key candidate in this query, because it appears in the target list, not the WHERE clause.

Note: The order of qualifications in the WHERE clause is not important, as the Ingres optimizer decides the appropriate order of execution.

Often, there are multiple candidate keys in a single query. Generally, the most restrictive column is the best key. The following example illustrates this:

```
Select empno from employee
  Where employee.sex = 'F'
  And employee.salary  > 20000'>
  And employee.name like 'Shigi%';
```

In this case, there are three columns that are potential keys. However, these first two qualifications are not very restrictive because “M” and “F” are the only two values for the key sex, and many employees are likely to have the selected salary qualification:

```
employee.sex = 'F'
```

```
employee.salary > 20000
```

The most restrictive qualification is probably:

```
employee.name like 'Shigi%'
```

Thus, name is chosen as the key column. Once you find all rows with names beginning with Shigi, it takes little time to determine which of these rows are female and make more than 20000, because the number of rows you are looking at is only a small subset of the employee records.

Secondary Keys

When evaluating multiple queries, you find situations where one table needs more than one key. Secondary indexes (see page 194) can provide a secondary key and can be employed in these circumstances, but indexes must be used with discretion, as they add overhead to update, delete, and insert operations.

For example, perhaps the administration department decides empno is the appropriate key for the employee table, but the shipping department prefers address as the key column of the table. Secondary indexes can alleviate this problem, but you have to weigh factors, such as the number of times a particular query is executed, the acceptable response time for a query, the time of day the query is likely to be executed, and the importance of a query in the global view of the application.

In evaluating how to key the employee table, each query type is ranked as in the following example:

	Query	**Number Executed Per Day**	**Acceptable Response Time**	**Time of Day**
1	select * from employee where empno = 123; KEY = empno	2000	1 second	7-4
2	select name from employee order by empno; no key, but sorted by empno	1	2 hours	after 5
3	select salary from employee where name = 'Shigio'; KEY = name	20	30 sec	9-5
4	select name from employee where comment = 'Fire'; KEY = comment	1	30 sec	9-5

The most important query to key in this list is Query 1 because it is executed frequently, requires fast response, and is pivotal to the application. The key choice for employee table is the empno column.

Query 2 does not contain a restriction, so no key decision must be made. Also, this report can be run at night, so CPU time is not crucial. Therefore, B-tree on empno is a good choice of storage structure and key, because both Query 1 and Query 2 benefit.

Query 3 is important, but it is not executed as frequently, nor does it require as immediate a response. A secondary key on name is appropriate.

Query 4 is not executed frequently, and although the importance rating for this query was high, it is advantageous to either work out a different implementation strategy or discourage the user from using this query often. The comment field is particularly large and empty and, therefore, is not a good key choice. A separate fired table can be set up that lists the employees who had been fired that day; this table is joined to the employee table.

Secondary Indexes

Secondary indexes provide a mechanism for specifying an additional key to the base table.

For instance, assume that an employee table containing name (employee's name) and empno (employee number) columns is hashed on empno, but occasionally data must be retrieved based on the employee's name rather than the employee number. You can create a secondary index on the name column of the table.

Working with Indexes

You can perform the following basic operations on indexes:

- Create index objects
- View existing index objects, including the detailed properties of each individual object
- Drop index objects

  Indexes are dropped automatically when the base table is destroyed. Indexes are also dropped when the base table is modified, unless the Persistence option is specified for the index.

In SQL, you can accomplish these tasks using the CREATE INDEX, HELP INDEX, and DROP INDEX statements. For details, see the *SQL Reference Guide*.

In VDBA, use the Indexes branch for a particular table in the Database Object Manager window. For detailed steps, see the Procedures section of online help.

Implementation and Overhead of Secondary Indexes

Secondary indexes are actually tables that are automatically tied to the base table. Secondary indexes must be updated whenever the base table is changed, so they must be used sparingly. The user need not explicitly reference the secondary index for it to be used in a query. In fact, you cannot directly update a secondary index and probably never reference it. If the Ingres optimizer sees that an index is available to help solve the query, generally the index is used.

By default, secondary indexes are created as ISAM tables. You can change the storage structure of the index by modifying the secondary index once it is created, or by specifying another structure when you create the index.

In VDBA, you create indexes using the Create Indexes dialog and modify them using the Modify Index Structure dialog. For more information on modifying an existing index, see the chapter "Maintaining Storage Structures."

The following example shows the relationship of a secondary index to a base table:

```
Select * from xnameselect name,tid from employee
|name                  |tidp||name                 |tid |
|-------------------------- ||-------------------------|
|Aitken                |3072||Gregori              |   0|
|Blumberg              | 512||Sabel                |   1|
|Brodie                |3584||Blumberg             | 512|
|Cameron               |1024||Kay                  | 513|
|Clark                 |4096||Shigio               | 514|
|Curan                 |1536||Cameron              |1024|
|Curry                 |4608||Mandic               |1025|
|Giller                |2048||Stannich             |1026|
|Gordon                |5120||Curan                |1536|
|Green                 |2560||McTigue              |1537|
|Gregori               |   0||Stover               |1538|
|Huber                 |3073||Giller               |2048|
|Kay                   | 513||Ramos                |2049|
|Kreseski              |3585||Verducci             |2050|
|Mandic                |1025||Green                |2560|
|McShane               |4097||Ross                 |2561|
|McTigue               |1537||Aitken               |3072|
|Ming                  |4609||Huber                |3073|
|Ramos                 |2049||Saxena               |3074|
|Robinson              |5121||Brodie               |3584|
|Ross                  |2561||Kreseski             |3585|
|Sabel                 |   1||Smith                |3586|
|Saxena                |3074||Clark                |4096|
|Shigio                | 514||McShane              |4097|
|Smith                 |3586||Stein                |4098|
|Stannich              |1026||Curry                |4608|
|Stein                 |4098||Ming                 |4609|
|Stover                |1538||Sullivan             |4610|
|Sullivan              |4610||Gordon               |5120|
|Verducci              |2050||Robinson             |5121|
|Zimmerman             |5122||Zimmerman            |5122|
|---------------------------||-------------------------|
```

There is a row in the secondary index xname for every row in the employee table. There is also a column called tidp. This is the *tid* of the row in the base table. Tids identify every row on every data page. For a complete discussion of tids, see Tids (see page 202). The tidp entry for an employee is the tid of the employee's record in the base table.

There are no limits to the number of secondary indexes that can be created on a table. However, there is overhead involved in the maintenance and use of a secondary index that you must be aware of:

- When you add a row to the base table, you add an entry into every secondary index on the table as well.
- When a row in the base table moves, causing the tid to change, every secondary index must be updated to reflect this change. In a base table, rows move when the key is updated or if the table is compressed and a row is replaced that no longer fits in the same page.

Note: For a compressed table, when a varchar(width) column is updated and then recompressed, the row size can change.

- When the base table is updated, so that there is a change of the value in a column, which is used as the key of a secondary index, the key of the secondary index has to be updated as well.
- When processing a query execution plan for a query, the more indexes and plans possible for the query, the longer it takes to decide what query execution plan to use.

R-tree Secondary Index

An R-tree storage structure is a secondary index for multi-dimensional object management extension data types that can provide the requisite functions (nbr and hilbert).

The R-tree index is a secondary index only. The access method of the base table is B-tree, hash, heap, or ISAM. The R-tree index uses two functions to describe and sort its data. The R-tree index is built on the nbr (normalized bounding rectangle) function of the original object, not the object itself. The nbr function describes the location of each object. The hilbert function sorts the nbr values so that nbr records describing close locations are close to one another in the R-tree index table.

For more information on the nbr and hilbert functions and for more information on handling objects, see the *Object Management Extension User Guide*.

An R-tree index allows Ingres to answer range queries, such as: “find all records where its position overlaps this spot,” quickly. Without an R-tree index, the whole database must be read. Consider two tables: Table A is a table of houses, and Table B contains park information and location. The query, “select all houses where the house intersects a park” is an example of a spatial join. Without an R-tree index, the spatial join reads Table B entirely for each row in Table A.

When creating an R-tree index (for example, using the Create Indexes dialog or the CREATE INDEX statement) you must include range values, which specify the minimum and maximum values of the index column.

The following example illustrates an R-tree index:

```
select shape, hex(hilbert), tidp from xfio_shape_ix;

+-------------------------------------------------+------------+-------+
|shape                                            |col2        |tidp   |
+-------------------------------------------------+------------+-------+
|((6644550,2412235),(6651911,2425562))            |182343433792|      0|
|((5711593,7469490),(5720615,7473074))            |2CBBAFC085E6|   1541|
|((5755540,7431379),(5765798,7468084))            |2CBC38CC815C|   1543|
|((5764642,7468084),(5776333,7489652))            |2CCEABAE4E25|   1542|
|((5760044,7471142),(5775065,7492024))            |2CCEAC433EF1|   1544|
|((4392392,7367220),(4392773,7368251))            |2F0514CC452B|      3|
|((4393222,7381338),(4393696,7382470))            |2F05ECE43CA5|   1536|
|((6105365,8716914),(6119516,8719411))            |7BC8B02F74CE|   1539|
|((6104208,8719411),(6123227,8733088))            |7BC8B47DB378|   1538|
|((6082882,8707086),(6104747,8708099))            |7BCA043955D6|   1540|
|((8995748,12135179),(8999981,12144160))          |8F8235359771|   1537|
|((9289826,13632441),(9325335,13663808))          |9356B03B9AA0|      1|
|((9268185,13666317),(9286628,13724240))          |93591514F7A8|      4|
|((9396304,16145868),(9397279,16148181))          |95C328081C95|      2|
|((11623892,4873084),(11624345,4874079))          |DF6722ADDB47|      7|
|((11624186,4871079),(11624855,4871713))          |DF6727B0C6D0|      6|
|((11622165,4875404),(11624949,4877801))          |DF672D336FDD|      8|
|((11621206,4874079),(11624345,4876640))          |DF672D738440|     10|
|((11621807,4874417),(11624499,4877759))          |DF672D7B50C1|      9|
|((11610646,4875871),(11612145,4878603))          |DF67321EEFB6|      5|
+-------------------------------------------------+------------+-------+
(20 rows)
```

The shape column contains the nbr coordinates. The col2 column contains the hilbert number for the nbr. The tidp column corresponds to the tid value of the object in the base table. Tids (see page 202) identify every row on every data page.

Secondary Indexes and Performance

Secondary indexes are generally used to index into the base table they see, although if the query can be executed in the secondary index alone, the base table need not be visited. Using secondary indexes to help complete queries that are otherwise executed on the base table can dramatically reduce the query execution time.

For example, assume a secondary index exists on the name column for the employee table, and the following query is executed:

```
select empno, age, name
  from employee
  where name like 'A%';
```

First, records beginning with an "A" in the secondary index are located, and using the tidp column, each tidp is used to do a tid lookup into the employee table, to get the rest of the information about the employee, namely empno and age. Tids (see page 202) identify every row on every data page.

Both the secondary index and the base table are used in this query. However, had the retrieval asked only for employee.name rather than empno and age, the base table is not used, and the number of disk I/Os executed is reduced by more than 50%.

Even in some situations requiring scans of the entire table, you can dramatically improve performance by loading the columns retrieved into the secondary index, so that probing the base table is not necessary. An example is shown in Example: Loading Retrieved Columns into a Secondary Index to Improve Performance (see page 200).

Example: Load Retrieved Columns into a Secondary Index to Improve Performance

In this example, the table bigtable contains 100,000 rows and 20,000 pages.

First, follow these steps to modify the bigtable to use a B-tree structure keyed on three columns:

1. In VDBA, open the Modify Table Structure dialog for bigtable. For more information, see the chapter "Maintaining Storage Structures" and online help.
2. Enable Change Storage Structure and click Structure.

   The Structure of Table dialog opens.
3. Select B-tree in the Structure drop-down list, enable col1, col2, and col3 in the Columns group box to specify them as keys, and then click OK.

   The Structure of Table dialog closes.
4. Click OK

   The Modify Table Structure dialog closes.

Next, a SELECT statement is issued in which the key columns are specified in the WHERE clause. This search requires a full table scan, even though the three columns in question are key columns in the bigtable structure:

```
select col1, col2, col3 from bigtable
  where col1 = 'Colorado', col2 = 17, col3 = 'repo';
```

Creating a secondary index on the three columns alleviates this problem.

Follow these steps to create a secondary index, with name xbig:

1. In VDBA, open the Create Indexes dialog for bigtable. For more information, see online help.
2. Enter **xbig** in the Index Name edit control.
3. For each of the key columns, col1, col2, and col3, select the column in the Base Table Columns list box, and click the double-right arrow (>>) to add it to the Index Columns list box, and then click OK.

The index xbig is 500 pages. Issuing the exact same query as before (shown again below) now uses the secondary index, thereby reducing the scan from 20,000 pages to 500 pages:

```
select col1, col2, col3 from bigtable
  where col1 = 'Colorado', col2 = 17, col3 = 'repo';
```

Aggregates on secondary indexes can be more efficient, because the index is so much smaller than the base table. For example, if there was a secondary index on col1, this aggregate is processed in much less time:

```
select avg(col1) from bigtable;
```

Forced Use of Secondary Indexes

You can force a secondary index to be used by referencing it in the query, but the optimizer must ensure that this is never necessary. For example, consider the following query:

```
select * from emp
  where emp.name = 'Shigio';
```

To force it to use a secondary index, change it to the following:

```
select * from emp, xname
  where xname.tidp = emp.tid
  and xname.name = 'Shigio';
```

Two Secondary Indexes

There is no reason for having two secondary indexes on the same column, for example, one hash and one ISAM. Instead, use the index giving you the most versatile access path because the overhead of maintaining and using two indexes is more than the disk I/O saved for a few queries.

If you need two access paths, and you want one to be hash and the other to be ISAM or B-tree, you can use ISAM (or B-tree) for the base table access method and hash for the index. ISAM and B-tree cluster similar data on the same data page, while hash randomizes data, so that ranges of values are not clustered. With the base table as ISAM or B-tree, range retrievals find the physical rows clustered on the same data pages, reducing the amount of disk I/O needed to execute range queries. If the base table is hash, the ISAM index points to the qualifying rows, but these rows are spread randomly about the table instead of being clustered on the same data pages.

Tids

Every row on every data page is uniquely identified by its page and row, known as its *tid*, or tuple identifier. Tids are designed to be used internally by the data manager. They are not supported for use in user-written programs.

Note: Tids were not designed to provide unique row identifiers for user data or to provide quick access. Tids are not stored, but are only calculated addresses, so they are unreliable row markers and are likely to change; in short, they must not be used by user programs or queries. We advise that you use tids for informational debugging purposes only. For more information, see the chapter "Understanding the Locking System."

Tids can be used for direct access into tables. B-tree leaf pages use tids to locate rows on data pages. Also, secondary indexes use tids to indicate which row the key value is associated with in the base table. When a secondary index is used to access a base table, the tid found in the tidp column is used to locate the row immediately in the base table. (The tidp column corresponds to the tid value of the object in the base table.) The base table's index structure is ignored and access is directly to the page and row.

A listing of the tids in the employee table illustrates tid numbering. The employee table is 500 bytes wide, so four rows fit on each 2048-byte data page.

Tid values start at 0 and jump by 512 each time a new page is encountered. In a page, each row is sequentially numbered:

0, 1, 2, 3, 512, 513, 514, 515, 1024, 1025, and so on.

For example, the relationship of tids to empno's in the employee table is illustrated as follows:

```
|empno                  |tid|
|---------------------------|
|            1|            0| Page 0  Row 0
|            2|            1|         Row 1
|            3|            2|         Row 2
|            4|            3|         Row 3
|            5|          512| Page 1  Row 0
|            6|          513|         Row 1
|            7|          514|         Row 2
|            8|          515|         Row 3
|            9|         1024| Page 2  etc.
|           10|         1025|
|           11|         1026|
|           12|         1027|
|           13|         1536| Page 3
```

Tids are not stored as part of each row; they are calculated as the data page is accessed.

If overflow pages are encountered, the tid values increase by more than 512; after the overflow chain, they again decrease. Overflow chains are particular to main data pages; however, they are always allocated at the end of the file as they are needed.

To illustrate overflow, assume the employee table was hashed with maxpages = 5. Given the following MODIFY and SELECT statements, the tid numbering is as shown here:

```
modify emp to hash on empno
    where maxpages = 5;
select name, tid from emp;

|name                          |tid|
|---------------------------------|
|Clark                        |   0| Page 0
|Green                        |   1|
|Mandic                       |   2|
|Robinson                     |   3|
|Smith                        |2560| OVERFLOW for Page 0
|Verducci                     |2561|
|Brodie                       | 512| Page 1
|Giller                       | 513|
|Kay                          | 514|
|Ming                         | 515|
|Saxena                       |3072| OVERFLOW for Page 1
```

Every tid value is unique. When a table is heap, tids always increase in value, because the pages always follow each other. B-tree data pages are not accessed directly, so tid values are not accessed sequentially (data is always sorted by key).

Tid values change as rows move; if a compressed row is expanded, its tid can change; if a key value is updated, the row is moved and the row’s tid changes. Although tids are retrievable, their values are unreliable in application programs. Use tids only to help to understand the structure of tables.

Chapter 10: Maintaining Storage Structures

This section contains the following topics:

Storage Structures and Performance

A major responsibility of the database administrator is to maintain good performance. Performance-enhancing tasks related to storage structures include:

- Modifying the database tables
- Compressing storage structures
- Managing overflow

You should understand when and how to use the modify procedures to change storage structures for tables and secondary indexes. As part of regular system maintenance, you should use modify procedures to eliminate overflow pages and recover disk space for deleted rows.

For additional information on database performance, see the chapter "Improving Database and Query Performance."

Table Pages

The data for each table is stored in a file on disk. Tables consist of pages with a size that you define when you create the table. For example, you can specify a page size of 2 KB, 4 KB, and so forth by powers of two up to 64 KB. Each page has a certain amount of overhead, which depends on the page size. Relevant values and how they are calculated for each possible page size are described in Space Requirements for Tables (see page 399).

Each page stores a number of rows. The number of rows per page varies, according to the row width, the storage structure of the table, whether or not the table is compressed, and how much data has been added or deleted because the table was last modified. Rows cannot span pages, limiting the maximum row width to the per-page data size.

The page is an important concept in understanding query performance because it affects the amount of disk I/O a query does, as well as the amount of CPU resources required to read through a table.

Display the Number of Pages in a Table

To see how many pages are in a table, you can use either VDBA or an SQL statement.

In VDBA, select a table and select the Pages tab.

In SQL, use the help table statement. For more information, see the *SQL Reference Guide*.

A display for a B-tree table is shown in this example:

```
Name:                   emp
Owner:                  ingres
Created:                22-sep-2006 10:27:00
Location:               ii_database
Type:                   user table
Version:                II9.0
Page size:              2048
Cache priority:         0
Alter table version:    0
Alter table totwidth:   70
Row width:              70
Number of rows:         32
Storage structure:      B-tree
Compression:            none
Duplicate Rows:         not allowed
Number of pages:        6
Overflow data pages:    0
Journaling:             enabled
Base table for view:    yes
Permissions:            none
Integrities:            none
Optimizer statistics:   none
Column Information:
                                                  Key
Column Name    Type     Length  Nulls   Defaults  Seq
name           varchar  20      no      no        1
title          varchar  15      no      yes
hourly_rate    money            no      yes
manager        varchar  20      yes     null

Secondary indexes:  none
```

Limitations of Heap Structure

Without help from the storage structure, when you want to retrieve a particular row from a table, you must search through every row to see if it qualifies. (Searching through every row is called *scanning* the table.) Stopping at the first row that qualifies is not enough, because multiple rows can qualify.

Consider the data shown in a sample heap table:

```
        empno  name        age   salary    comment
        +-------------------------------------------
Page 0  | 17 | Shigio    |  29| 28000.000|
        |  9 | Blumberg  |  33| 32000.000|
        | 26 | Stover    |  38| 35000.000|
        |  1 | Mandic    |  46| 43000.000|
        |-------------------------------------------
Page 1  | 18 | Giller    |  47| 46000.000|
        | 10 | Ming      |  23| 22000.000|
        | 27 | Curry     |  34| 32000.000|
        |  2 | Ross      |  50| 55000.000|
        |-------------------------------------------
Page 2  | 19 | McTigue   |  44| 41000.000|
        | 11 | Robinson  |  64| 80000.000|
        | 28 | Kay       |  41| 38000.000|
        |  3 | Stein     |  44| 40000.000|
        |-------------------------------------------
Page 3  | 20 | Cameron   |  37| 35000.000|
        | 12 | Saxena    |  24| 22000.000|
        | 29 | Ramos     |  31| 30000.000|
        |  4 | Stannich  |  36| 33000.000|
        |-------------------------------------------
Page 4  | 21 | Huber     |  35| 32000.000|
        | 13 | Clark     |  43| 40000.000|
        | 30 | Brodie    |  42| 40000.000|
        |  5 | Verducci  |  55| 55000.000|
        |-------------------------------------------
Page 5  | 22 | Zimmerman |  26| 25000.000|
        | 14 | Kreseski  |  25| 24000.000|
        | 31 | Smith     |  20| 10000.000|
        |  6 | Aitken    |  49| 50000.000|
        |-------------------------------------------
Page 6  | 23 | Gordon    |  28| 27000.000|
        | 15 | Green     |  27| 26000.000|
        |  7 | Curan     |  30| 30000.000|Fire
        | 24 | Sabel     |  21| 21000.000|
        |-------------------------------------------
Page 7  | 16 | Gregori   |  32| 31000.000|
        |  8 | McShane   |  22| 22000.000|
        | 25 | Sullivan  |  38| 35000.000|
        |
        +-------------------------------------------
```

With this heap structure, a retrieval such as the following looks at every page in the emp table:

```
select * from emp where emp.name = 'Sullivan';
```

Although the Shigio record is the first row in the table, the following retrieval also looks at every row in the table:

```
select * from emp where emp.name = 'Shigio';
```

Because the table is not sorted, the entire table must be scanned in case there is another employee named Shigio on another page in the table.

Retrieval from a large table can be costly in time and system resources. To understand the performance consequences of a scan of a large table, assume that the emp table is actually 300,000 pages, rather than 8. Further, assume the disks can manage approximately 30 disk I/Os per second. Assume one disk I/O per page. With a heap storage structure, the example select operation takes 300,000 / 30 = 10,000 seconds (or 2 hours, 46 minutes) in disk access time alone, not counting the CPU time taken to scan each page once it is brought in from disk, and assuming no other system activity.

For a large table, a different storage structure is needed. A production system cannot tolerate a three-hour wait to retrieve a row. The solution is to provide a storage structure that allows for keyed access, like hash, ISAM, or B-tree.

Modify Procedures

To improve performance, you can change tables to a more effective storage structure by using modify procedures.

Key Columns and Performance

For hash, ISAM, and B-tree structures, you must specify key columns. (Heap and heapsort tables do not have key columns.) There is no limit to the number of key columns that can be specified, but as key columns increase, performance declines slightly.

Tools for Modifying Storage Structures

In VDBA, to change a table from one storage structure to another, use the Modify Table Structure dialog. By enabling the Change Storage Structure radio button and clicking Structure, you activate the Structure of Table dialog, where you can specify the parameters for the storage structure type and other structure-specific characteristics. For secondary indexes, the Modify Index Structure dialog offers a similar option to enable the Structure of Index dialog. For more information, see Modifying Storage Structures in online help.

Using SQL, you can accomplish this task with the MODIFY statement. For more information, see the *SQL Reference Guide*.

Cautions When Using the Modify Procedure

Keep in mind the following effects of the modify procedure when you are modifying the storage structure:

- Locking—During the modify procedure, the table is exclusively locked and inaccessible to other users.
- Secondary Indexes—Secondary indexes are destroyed when you modify the base table storage structure. Modifying Secondary Indexes (see page 226) provides more information.
- Disk Space—When a table storage structure is modified, temporary sort files are created. Before the old table can be deleted, a new table must be built. Once it is completely built, the old table is deleted, and the temporary file is renamed with the old table name. Space Requirements for Modify Operations (see page 406) provides more information.
- Partitioned tables—Modifying a table with a large number of partitions requires a large amount of space in the transaction log file. It is possible to fill the log file with a modify of a partitioned table.

Options to the Modify Procedure

The modify procedure provides several options:

- Min Pages
- Max Pages
- Allocation
- Extend
- Fillfactor
- Leaffill
- Nonleaffill
- Unique
- Compression

The MinPages, MaxPages, Allocation, Fillfactor, Leaffill, and Nonleaffill options take effect during the modify procedure only, but are remembered in the system catalog description of the table. They will be applied again by a future modify-to-reconstruct, and will be output as part of the table description by copydb and unloaddb. The Extend, Unique, and Compression options are continuously active throughout the life of the table.

In VDBA, these options are in the Structure of Table and Structure of Index dialogs.

Number of Pages

Min Pages and Max Pages are valid options only when you are modifying the table to hash. These options allow you to control the hashing algorithm to some extent, extending the control offered by the Fillfactor option.

The Min Pages option is useful if the table will be growing rapidly or if you want few rows per page to increase concurrency so multiple people can update the same table.

You can achieve nearly the same effect by specifying a low value for the Fillfactor option, but the fill factor is based on the current size of the table, as described in Alternate Fill Factors (see page 216).

To force a specific number of main pages, use the Min Pages option to specify a minimum number of main pages. The number of main pages used are at least as many as specified, although the exact number of Min Pages specified is not used.

Example: Modify Structure and Force a Higher Number of Main Pages for a Table

For example, for the emp table in the previous chapter you can force a higher number of main pages by specifying the minimum number of main pages when you modify the table to hash. If you specify 30 main pages for the table, which has 31 rows, you have approximately one row per page.

Follow these steps to modify the storage structure of the emp table:

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select Hash from the Structure drop-down list.
3. Enter **30** in the Min Pages edit control.
4. Enable the age column in the Columns list.

To specify a maximum number of main pages to use, rather than the system choice, use the Max Pages option. If the number of rows does not completely fit on the number of pages specified, overflow pages are allocated. If fewer pages are needed, the lesser number is used. Max Pages is useful mainly for shrinking compressed hash tables more than otherwise happens.

You can achieve nearly the same effect by specifying a high value for the Fillfactor option, but the fill factor is based on the current size of the table, as described in Alternate Fill Factors (see page 216).

Example: Specify a Maximum Number of Main Pages for a Table

The following example modifies the emp table, specifying a Max Pages value.

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select Hash from the Structure drop-down list.
3. Enter **100** in the Max Pages edit control.
4. Enable the empno column in the Columns list.

Remember that Max Pages controls only the number of main pages; it does not affect overflow pages. For example, assume your data takes 100 pages in heap. If you modify the table to hash and limit the number of main pages to 50, the remainder of the data goes onto overflow pages.

Allocation of Space

Use the Allocation option to pre-allocate space. You can modify the table to an allocation greater than its current size to leave free space in the table. (The default is four pages if no allocation has been specified.)

Doing this allows you to avoid a failure due to lack of disk space, or to provide enough space for table expansion instead of having to perform a table extend operation. Extending a Table or Index (see page 226) provides more information.

The allocated size must be in the range 4 to 8,388,607 (the maximum number of pages in a table). The specified size is rounded up, if necessary, to make sure the allocation size for a multi-location table or index is always a multiple of sixteen.

Note: If the specified number of pages cannot be allocated, the modify procedure is aborted.

After an allocation is specified, it remains in effect and does not need to be specified again when the table or index is modified.

Example: Allocate 1000 Pages to a Table

The following example specifies that 1000 pages be allocated to table inventory:

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select B-tree from the Structure drop-down list.
3. Enter **1000** in the Allocation edit control.

   The space allocated is 1008, due to rounding.

Extension of Space

The Extend option allows you to control the amount of space by which a table is extended when more space is required. (The default extension size is 16 pages.)

The size must be in the range 1 to *max_size*, where the *max_size* is calculated as:
8,388,607 – allocation_size.

The specified Extend size is rounded up, if necessary, to make sure the size for a multi-location table or index is always a multiple of sixteen.

Note: If the specified number of pages cannot be allocated, the operation fails with an error.

After an extend size has been specified for the table or index, it remains in effect and does not need to be specified again when the table or index is modified.

Example: Extend a Table in Blocks of 1000 Pages

The following example specifies that the table inventory be extended in blocks of 1000 pages:

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select B-tree from the Structure drop-down list.
3. Enter **1000** in the Extend edit control.

   The extension space is 1008, due to rounding.

Guidelines for Choosing an Extend Size

When choosing an extend size, keep the following in mind:

- When extending a table, not only the physical extension must be performed, but the extension must also be recorded. Therefore, avoid an excessively small extend size that requires many additional small extensions.
- In an environment that is short of disk space, a large extend size can cause an operation to fail, even when there is sufficient disk space for the particular operation.
- **Windows:** On a file system that requires the underlying files to be written to when allocating disk space, a large extend size can be undesirable because it affects the performance of the operation that causes the extension.
- **UNIX:** On a file system that requires the underlying files to be written to when allocating disk space, a large extend size can be undesirable because it affects the performance of the operation that causes the extension.
- **VMS:** On file systems that provide calls for allocating disk space, a large extend size helps reduce the amount of table fragmentation.

Default Fill Factors

Each storage structure has a different default fill factor. The term *fill factor* refers to the number of rows that are actually put on a data page divided by the number of rows that fit on a data page for a particular structure.

The various fill factors enable you to add data to the table without running into overflow problems. Because the data pages have room to add data, you do not have to remodify.

For instance, a heap table fits as many rows as possible on a page; this is known as 100% fill factor. However, ISAM and B-tree data pages are filled only to 80% capacity, leaving room to add 20% more data before a page is completely full.

The default data page fill factors are as follows:

Storage Structure	Default Fill Factor	Multiply Heap Size by	Number of Pages Needed for 100 Full Pages
B-tree	80%	1.25	125 + index pages
compressed B-tree	100%	1	100 + index pages
hash	50%	2	200
compressed hash	75%	1.34	134
heap	100%	1	100
compressed heap	100%	1	100
ISAM	80%	1.25	125 + index pages
compressed ISAM	100%	1	100 + index pages

The default B-tree index page fill factors are as follows:

Storage Structure	Default Fill Factor
B-tree leaf	70%
B-tree index	80%

The first table shows that if a heap table is 100 pages and you modify that table to hash, the table now takes up 200 pages, because each page is only 50% full.

Note: Depending on the system allocation for tracking used and free pages, the number of pages can be approximate. For more information, see the chapter "Calculating Disk Space."

Alternate Fill Factors

You can tailor the fill factor for various situations. For instance, if the table is not going to grow at all, use a 100% fill factor for the table. On the other hand, if you know you are going to be adding a lot of data, you can use a low fill factor, perhaps 25%. Also, if your environment is one where updates are occurring all the time and good concurrency is important, you can set the fill factor low.

Note: Fill factor is used only at modify time. As you add data, the pages fill up and the fill factor no longer applies.

When specifying a fill factor other than the default, you must keep the following points in mind:

- Use a high fill factor when the table is static and you are not going to be appending many rows.
- Use a low fill factor when the table is going to be growing rapidly. Also, use a low fill factor to reduce locking contention and improve concurrency. A low fill factor distributes fewer keys per page, so that page level locks lock fewer records.

Specifying fill factor is useful for hash and ISAM tables. However, for B-tree tables, because data pages only are affected, the Fillfactor option must be used with the Leaffill or Nonleaffill options. See Leaf Page Fill Factors (see page 218) and Index Page Fill Factors (see page 219).

For hash tables, typically a 50% fill factor is used for uncompressed tables. You can raise or lower this, but raising it too high can cause more overflow pages than desirable. You must always measure the overflow in a hash table when setting a high fill factor—fill factors higher than 90% are likely to cause overflow.

If you are using compressed ISAM tables and are adding data, make sure you set the fill factor to something lower than the default 100%, or you immediately add overflow pages.

Normally, uncompressed ISAM tables are built with an 80% fill factor. You can set the fill factor on ISAM tables to 100%, and unless you have duplicate keys, you cannot have overflow problems until after you add data to the table.

In VDBA, you control the fill factor of the data pages using the Fillfactor option in the Structure of Table and Structure of Index dialogs.

Example: Set Fill Factor to 25% on a Hash Table

This example sets the fill factor on a hash table to 25%, rather than the default of 50%, by modifying the emp table:

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select Hash from the Structure drop-down list.
3. Enter **25** in the Fillfactor edit control.
4. Enable the empno column in the Columns list.

Example: Set Fill Factor to 100% on an Uncompressed ISAM Table

This example sets the fill factor on an uncompressed ISAM table to 100%:

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select Isam from the Structure drop-down list.
3. Enter **100** in the Fillfactor edit control.
4. Enable the name column in the Columns list.

Leaf Page Fill Factors

It is possible to specify B-tree leaf page fill factors at modify time. This is the percentage of the leaf page that is used during the modify procedure. The remaining portion of the page is available for use later when new rows are added to the table.

The purpose of the fill factor is to leave extra room on the leaf pages to do inserts without causing leaf page splits. This is useful if you modify a table to B-tree and plan to add rows to it later.

In VDBA, you control these values using the Leaffill options in the Structure of Table dialog.

The Leaffill option specifies the percentage of each leaf page to be filled at the time the table is modified to B-tree or cB-tree. The Leaffill default is 70, which means that 70% of the leaf page is filled at modify time and 30% remains empty for future use.

For example, assume that the key-tid pair requires 400 bytes of storage. This means that five key-tid pairs fit on a single 2 KB B-tree leaf page. However, if the leaf page fill factor is specified at 60%, only three key-tid pairs are allocated on each B-tree leaf page at modify time. If subsequent updates to the table cause two new rows on this leaf page, they are placed in the empty space on the leaf page. The key-tid pairs are reordered on the leaf page from min to max. If more than two new rows need to be added to this leaf page, there is not enough space and the leaf page has to split.

Index Page Fill Factors

It is possible to specify B-tree index page fill factors at modify time. This is the percentage of the index page that is used during the modify procedure. The remaining portion of the page is available for use later when new rows are added to the table. The purpose of the fill factor is to leave extra room on the index pages to do inserts without causing index page splits. This is useful if you modify a table to B-tree and plan to add rows to it later.

In VDBA, you control these values using the Nonleaffill options in the Structure of Index dialog.

The Nonleaffill option specifies the percentage of each index page that is to be filled at the time the table is modified to B-tree. That is, it is similar to Leaffill, but for index pages instead of leaf pages. The Nonleaffill default is 80. This means that 80% of the index page is used at modify time and 20% remains empty for future use.

For example, assume that the key-tid pair requires 500 bytes of storage. This means that four key-tid pairs fit on a single B-tree index page. However, if the index page fill factor is specified at 75%, only three key-tid pairs are allocated on each 2 KB B-tree index page at modify time. If subsequent updates to the table cause another leaf page to be allocated, the empty space on the index page is used to hold a key-tid pair for that new leaf page. If there are enough new rows to cause two new leaf pages to be added to that index page, the index page must split. For more information, see Tids (see page 202).

Setting a fill factor of lower than 60 on leaf pages can help reduce locking contention when B-tree leaf pages are splitting, because index splitting is reduced. Setting Leaffill low for small but quickly growing B-trees is advisable.

When you specify a high Leaffill, index splitting is almost guaranteed to occur because leaf pages immediately fill up when data is added. Thus, you want to avoid a high fill factor unless the B-tree table is relatively static. Even in this case, use an ISAM table.

Ensuring Key Values Are Unique

Unique keys can be enforced automatically for hash, ISAM, and B-tree tables using the modify procedure.

Benefits of Unique Keys

Benefits of unique keys are:

- A good database design that provides unique keys enhances performance.
- You are automatically ensured that all data added to the table has unique keys.
- The Ingres optimizer recognizes tables that have unique keys and uses this information to plan queries wisely.

In most cases unique keys are an advantage in your data organization.

Disadvantages of Unique Keys

The disadvantages of unique keys include a small performance impact in maintaining uniqueness. You must also plan your table use so that you do not add two rows with the same key value.

Specify Unique Keys

In VDBA, unique keys can be specified as Row or Statement in the Unique group box in the Structure of Table and Structure of Index dialogs:

- Row indicates that uniqueness is checked as each row is inserted.
- Statement indicates that uniqueness is checked after the update statement is executed.

If you do not want to create a unique key, select the No option.

Example: Prevent the Addition of Two Names with the Same Number

The following example prevents the addition of two employees in the emp table with the same empno:

1. In VDBA, open the Structure of Table dialog for the emp table. For more information, see online help.
2. Select Isam from the Structure drop-down list.
3. Enable Row in the Unique radio button group box.
4. Enable the empno column in the Columns list.

If a new employee is added with the same employee number as an existing record in the table, the row is not added, and you are returned a row count of zero.

Note: An error is not returned in this case; only the row count shows that the row was not added. Be aware of this if you are writing application programs using unique keys.

Example: Modify a Table to Hash and Prevent the Addition of Two Names with the Same Number

The following example modifies the emp table to hash and prevents the addition of two employees in the emp table with the same empno.

1. In VDBA, open the Structure of Table dialog for the emp table. For more information, see online help.
2. Select Hash from the Structure drop-down list.
3. Enable Row in the Unique radio button group box.
4. Enable the empno column in the Columns list.

The rows in the following example have unique keys. Although employee #17 and #18 have the same records except for their employee numbers, the employee numbers are unique, so these are valid rows after the modification:

```
    Empno  Name   Age  Salary
   | 17 | Shigio | 29| 28000.000|
   | 18 | Shigio | 29| 28000.000|
   |  1 | Aitken | 35| 50000.000|
```

The following two rows do not have unique keys. These two rows cannot both exist in the emp table after modification to hash unique on empno:

```
    Empno  Name   Age  Salary
   | 17 | Shigio | 29| 28000.000|
   | 17 | Aitken | 35| 50000.000|
```

Table Compression

All storage structures—except R-tree secondary index and heapsort—permit tables and indexes (where present) to be compressed.

Compression is controlled using the Key and Data options in the Compression group box in the Structure of Table and Structure of Index dialogs. By default, there is no compression when creating or modifying.

Not all parts of all storage structures can be compressed, as summarized in the table below:

Storage Structure		**Data**	**Key**
B-tree	Base Table	Yes	Yes
	Secondary Index	No	Yes
hash	Base Table	Yes	No
	Secondary Index	Yes	No

Storage Structure		Data	Key
heap	Base Table	Yes	No
	Secondary Index	N/A	N/A
heapsort	Base Table	No	No
	Secondary Index	N/A	N/A
ISAM	Base Table	Yes	No
	Secondary Index	Yes	No
R-tree	Base Table	N/A	N/A
	Secondary Index	No	No

Note: In VDBA, selecting Data in the Compression group box in the Structure of Table dialog does not affect keys stored in ISAM or B-tree index and leaf pages—only the data on the data pages is compressed. To compress index entries on B-tree index pages, select Key instead.

ISAM index pages cannot be compressed.

Compression of tables compresses character and text columns. Integer, floating point, date, and money columns are not compressed, unless they are nullable and have a null value.

Trailing blanks and nulls are compressed in character and text columns. For instance, the emp table contains a comment column that is 478 bytes. However, most employees have comments that are only 20 to 30 bytes in length. This makes the emp table a good candidate for compression because 478 bytes can be compressed into 30 bytes or fewer, saving nearly 450 bytes per row.

Furthermore, as many rows are placed on each page as possible, so that the entire emp table (31 rows) that normally took eight 2KB pages as a heap, takes just one page as a compressed heap. In this example, pages were limited to four rows per page, but by using compression, many more rows can be held per page.

There is no formula for estimating the number of rows per page in a compressed table, because it is entirely data dependent.

When to Compress a Table

When a table is compressed, you can reduce the amount of disk I/O needed to bring a set of rows from disk. This can increase performance if disk I/O is a query-processing bottleneck.

For instance, having compressed the emp table from eight pages down to one page, the following query performs only one disk I/O, whereas prior to compression as many as eight disk I/Os were required:

```
select * from emp;
```

In a large table, compression can dramatically reduce the number of disk I/Os performed to scan the table, and thus dramatically improve performance on scans of the entire table. Compression is also useful for conserving the amount of disk space it takes to store a table.

Compression Overhead

Compression must be used wisely, because the overhead associated with it can sometimes exceed the gains.

If a machine has a fast CPU, disk I/O can be the bottleneck for queries. However, because compression incurs CPU overhead, the benefits must be weighed against the costs, especially for machines with smaller CPUs. Compression can increase CPU usage for a query because data must be decompressed before it is returned to the user. This increase must be weighed against the benefits of decreased disk I/O and how heavily loaded the CPU is. High compression further reduces disk I/O, but uses even more CPU resources.

There is overhead when updating compressed tables. As rows are compressed to fit as many as possible per page, if you update a row so that it is now larger than it was before, it must be moved to a new spot on the page or even to a new page. If a row moves, its tid, or tuple identifier, also changes, requiring that every secondary index on the compressed table also be updated to reflect the new tid. For more information, see Tids (see page 202).

For example, if you change Shigio’s comment from “Good” to “Excellent,” Shigio’s record length grows from 4 bytes to 9 bytes and does not fit back in exactly the same place. His record needs to be moved to a new place (or page), with updates made to any secondary indexes of this table (if the emp table was B-tree, the appropriate B-tree leaf page is updated instead).

Compressed tables must be avoided when updates that increase the size of text or character columns occur frequently, especially if there are secondary indexes involved—unless you are prepared to incur this overhead. If you do compress and are planning to update, use a fill factor lower than 100% (75% for hash); the default fill factor for compressed tables is 75% for hash with data compression, 100% for the others. With free space on each page, moved rows are less likely to be placed on overflow pages. For more information, see Options to the Modify Procedure (see page 210).

Page Size

The default page size is 8 KB. The corresponding buffer cache for the installation must also be configured with the page size you specify or you receive an error. For more information, see the "Configuring Ingres" chapter in the *System Administrator Guide*.

For more information on page size see Table Pages (see page 206).

Shrinking a B-tree Index

To maintain good concurrency and performance, the B-tree index is not rebuilt after deletions. Deletions occur at the leaf and data page level, but an empty leaf page is not released. If your environment is one where many deletions are performed, you must occasionally update the index

In VDBA, you do this using the Shrink B-tree Index option in the Modify Table Structure and Modify Index Structure dialogs.

In SQL, you accomplish this task with the MODIFY statement. The TO MERGE clause is the same as the Shrink B-tree Index option. For more information, see the *SQL Reference Guide*.

The Shrink B-tree Index option is also important for users with incremental keys, which can incur lopsided indexes after heavy appends to the end of the table.

Not updating the index to reflect unused leaf pages can cause the index to be larger than necessary.

For example, if the emp table is keyed on empno (ranging from 1 to 31), and you fire all employees with employee numbers less than 16, the B-tree index does not shrink, but is unbalanced. This is shown in the following "Before" diagram:

Before

```
        <=16                 >16

       /                             \
     <=8      >8              <=24    >24
      /        \               /        \
<=4  >4   <=12  >2      <=20  >20  <=28  >28

Page 1     Page 2       Page 3       Page 4
(deleted   (deleted     valid        valid
 data)      data)       data         data
```

To re-balance the index level, you can use the Shrink B-tree Index option. It also reclaims unused leaf pages that otherwise are never reused. This is shown in the following "After" diagram:

After

```
          <= 24        >24

           /                  \
         <=16   >16      <=28   >28

         Page 3         Page 4
         valid          valid
         data           data

     Free page list: 1,2
```

The index is rebuilt, and empty leaf pages are marked as free, but otherwise leaf and data pages remain untouched. Therefore, this procedure is neither as time-consuming nor as disk-space intensive as modifying the table structure using the Change Storage Structure option. Shrink B-tree Index, however, does not re-sort the data on the data pages. Modifying the structure to B-tree is the only option for resorting data on data pages.

Extending a Table or Index

You can extend (add pages to) a table or index. You must specify the number of pages you want to add. Using this option does not rebuild the table or drop any secondary indexes.

In VDBA, you can extend a table or index by enabling the Add Pages radio button in the Modify Table Structure or Modify Index Structure dialogs and specifying the number of pages to add.

In SQL, you can accomplish this task with the MODIFY statement. The WITH EXTEND clause is the same as the Add Pages option. For more information, see the *SQL Reference Guide*.

Modifying Secondary Indexes

Secondary indexes are destroyed by default when you modify the base table storage structure. They are destroyed automatically because secondary indexes use the tidp column to reference the row of the base table to which they are pointing. When you modify a table, all the tids of the rows in the base table change, rendering the secondary index useless. For more information, see Tids (see page 202).

Persistence Option

You can use the Persistence option when creating or modifying a secondary index to specify that the index be recreated whenever the base table is modified. By default, indexes are created with no persistence.

In SQL, you can accomplish this task with the CREATE INDEX and MODIFY statements, and the [NO]PERSISTENCE clause. For more information, see the *SQL Reference Guide*.

In VDBA, this option is found in the Structure of Index and the Create Indexes dialogs.

Example: Enable the Persistence Option

For example, assuming the secondary index empidx was created without enabling the Persistence option, you can modify it to enable this feature, as follows:

1. In VDBA, open the Structure of Index dialog for the empidx index. For more information, see online help.
2. Select B-tree from the Structure drop-down list.
3. Enable the Persistence check box.

Changing the Index Storage Structure

The default storage structure for secondary indexes is ISAM; you can choose a different structure when creating an index.

To do this in VDBA, use the Create Indexes dialog.

You can also modify the index to another storage structure after it has been created.

To do this in VDBA, use the Structure of Index dialog.

If a secondary index is modified to B-tree, it cannot contain any data pages. Instead, the leaf pages in the secondary index point directly to data pages in the main table.

Overflow can occur in hash and ISAM secondary indexes, as well as base tables, and must be monitored. One way to eliminate overflow is to use B-tree as the default index structure. If overflow is not a problem, hash or ISAM can be preferable because the indexes are smaller, require less locking, and reuse deleted space.

Secondary indexes are smaller and can be modified more quickly than the base table. When they are used, overflow occurs less frequently because only key values are stored, rather than the entire row.

Because it is quicker to build secondary indexes than to modify the base table, it is easier to experiment with different choices of secondary indexes and different storage structures for them. Remember, however, that it can take longer to update a table with secondary indexes than one without them.

A high degree of duplication in a secondary key can lead to overflow in the secondary index. Repetitive keys are not recommended. Performance benefits can be derived by the inclusion of another column in the secondary index that makes the entire key less repetitive. The less repetitive key reduces the likelihood of overflow chains, resulting in better performance when updates made to the base table require updates to the secondary index. Because overflow chains are reduced, locking and searching overhead is lessened.

If the secondary index to be stored is ISAM or B-tree and the key is not unique, the tidp column is automatically included in the key specified when the index is modified. This achieves key uniqueness without any loss of functionality when the key is used for matches.

Example: Create a B-tree Index for a Table

The following example creates a B-tree index for the emp table:

1. In VDBA, open the Create Indexes dialog for the table. For more information, see the online help. Also see the chapter "Choosing Storage Structures and Secondary Indexes."
2. Enter an appropriate name in the Index Name edit control.
3. Select B-tree from the Structure drop-down list.
4. Select an appropriate key column in the Base Table Columns list box, and click the double-right arrow (>>) to add the column to the Index Columns list box.

Example: Modify an Existing Index to B-tree

This example modifies an existing index to use the B-tree storage structure (assuming it was created using another storage structure):

1. In VDBA, open the Structure of Index dialog for the index. For more information, see online help.
2. Select B-tree from the Structure drop-down list.
3. Enable the appropriate columns in the Columns list.

Remodifying B-tree Tables

If you suspect that the data on the data pages is scattered over several data pages, you can modify the table to B-tree again. You can check this by retrieving the tids as well as the column values, and looking at the pages they reflect.

Remodifying sorts the data and builds the B-tree index, placing like keys on the same data pages, which can slightly reduce the number of disk I/Os required to access the data. For more information, see Tids (see page 202).

This type of modification is especially useful when the key size is small, the row size is large, and the data has not been appended in sorted order. Remodifying a B-tree is also useful when you have deleted many rows and must reclaim disk space. For more information, see Tracking of Used and Free Pages (see page 404).

Examples: Remodifying a Table to B-tree

The first example represents the table before modification, and the second example shows it after modification.

The following retrieval touches all three data pages before modification but only one page after modification:

```
select * from emp where emp.age = 35;
```

The following table shows the leaf and data pages prior to modification. The records with a key of 35 are found on several data pages:

```
Leaf Page
key   page,row (tid)
35    1,2 (514)
35    2,2 (1026)
35    3,3 (1539)
36    2,3 (1027)
37    3,2 (1538)

Data Pages
Page 1         Page 2          Page 3
1,1 (513) 29   2,1 (1025) 29   3,1 (1537) 30
1,2 (514) 35   2,2 (1026) 35   3,2 (1538) 37
1,3 (515) 30   2,3 (1027) 36   3,3 (1539) 35
```

The following example modifies the emp table, respecifying B-tree as its structure.

1. In VDBA, open the Structure of Table dialog for the table. For more information, see online help.
2. Select B-tree from the Structure drop-down list.
3. Enable the age column in the Columns list.

After you perform this modification, the table looks as follows. All records with a key of 35 are clustered together on Page 2:

```
Page 1         Page 2          Page 3
1,1 (513) 29   2,1 (1025) 35   3,1 (1537) 36
1,2 (514) 29   2,2 (1026) 35   3,2 (1538) 37
1,3 (515) 30   2,3 (1027) 35
```

Common Errors During the Modify Procedure

When using the modify procedure, the most common errors include:

- A "duplicate key" error message when you use the Unique option (or the TO UNIQUE clause of the MODIFY statement).

  To resolve this problem, determine which rows have duplicate keys and delete these rows. You can locate these rows with the following query:

  ```
  select key_col, count(*) as repeat_number
    from table_name
    group by key_col
    having count(*) > 1;
  ```

- An error when modifying a table.

  You may be out of disk space on the file system the modify procedure is trying to use. Clear up disk space on this file system.

Overflow Management

Overflow chains can slow down performance considerably. Overflow must be monitored and prevented as much as possible.

Preventing or reducing overflow requires you to do the following:

- Carefully monitor overflow in both primary tables and secondary indexes
- Avoid the use of repetitive keys, including both primary keys and secondary index keys
- Modify table structure to redistribute poorly distributed overflow
- Understand the overflow implications when choosing a particular storage structure

Measure the Amount of Overflow

You can monitor overflow using either VDBA or an SQL statement.

In VDBA, select a table or secondary index in the Database Object Manager window, and click the Pages tab.

In SQL, you can monitor overflow with the help table statement. For more information, see the *SQL Reference Guide*.

For tables, overflow data is displayed in red in the pie chart, as indicated in the legend. Heap tables are considered as one main page, with an overflow chain attached to the main page. For B-tree tables, overflow occurs only at the leaf level and only with duplicate keys.

The iitables catalog (a view into the iirelation catalog) includes one row for each table in the database. It contains pertinent information for evaluating overflow.

For example, the following query results in the information shown in the table:

```
select table_name, storage_structure,
  number_pages, overflow_pages
  from iitables
```

table_name	storage_structure	number_pages	overflow_pages
manager	hash	22	4
department	B-tree	5	0
parts	B-tree	5	0
orders	heap	3	0

The above figures are approximate; they are updated only when they change by a certain percentage (5%) to prevent performance degradation by continuously updating these catalogs. Also, if transactions that involve many new pages are backed out during a recovery, the page counts cannot be updated. Page counts are guaranteed to be exact only after modification.

In evaluating overflow, if the number of overflow pages is greater than 10-15% of the number of data pages, expect performance degradation. Overflow must be regularly monitored to ensure that performance does not degrade as rows are appended to tables.

Repetitive Key Overflow

Storage structures other than heap that have a high degree of duplication in the key values are likely to have overflow because duplicate keys are stored in overflow pages. Keys with a high degree of duplication are not recommended. This applies to secondary index keys as well as primary keys.

Repetitive key overflow occurs, for example, if the emp table is keyed on sex, resulting in two primary pages for the values "M" and "F." The remainder of the pages are overflow pages to these two primary pages.

Consider if the following query is run:

```
select * from student
  where student.sex = 'F'
  and student.name = 'Baker';
```

The key is used to find the first primary page. The search goes down the entire overflow chain for "F" looking for all names Baker. Every page is checked. Because this query looks restrictive, the locking system probably chooses to page level lock. The query locks 10 pages and eventually escalates to a table level lock. Wait for the table level lock if other users are updating. Finally, the search finishes scanning the overflow chain and returns the row.

Retrieval performance with a duplicate key is still better than for a heap table because only half the table is scanned.

However, update performance suffers. If a user wants to append a new female student, the locking system starts by exclusively locking pages in the "F" overflow chain. If another 10 pages need to be locked eventually, the locking system attempts to escalate to an exclusive table level lock. If only one user is updating the table, the lock is easily obtained. If multiple users are trying to update the table at the same time, deadlock is likely.

User1 and User2 both exclusively hold 10 pages in the table. User1 wants to escalate to an exclusive table level lock so the query can continue, but User1 cannot proceed until User2 drops the exclusive page level locks User2 holds. User2 also wants to obtain an exclusive table level lock, but cannot proceed until User1 releases the locks. This is deadlock, which can seriously degrade update performance. For more information, see the chapter "Understanding the Locking System."

Poorly Distributed Overflow

Overflow that is not uniformly distributed, that is, it is concentrated around one or two primary pages, is poorly distributed. A classic example of poorly distributed overflow occurs when new rows are added to a table with a key that is greater than all the keys that already exist in the table (for example, a time stamp). If this table has an ISAM structure, the table builds up overflow in the last primary page, and all operations involving this overflow chain can exhibit poor performance. This type of table is best stored as a B-tree or hash.

Overflow and ISAM and Hash Tables

In hash and ISAM tables that have had a large amount of data added and have not been remodified, overflow and the resulting performance degradation is easy to understand. A keyed retrieval that normally touches one page now has to look through not only the main data page, but also every overflow page associated with the main data page. For every retrieval, the amount of disk I/O increases as the number of overflow pages increases.

Overflow pages are particular to a main data page for ISAM and hash tables, not to the table itself. If a table has 100 main pages and 100 overflow pages, it is likely that the overflow pages are distributed over many main data pages (that is, each main data page has perhaps one overflow page). A keyed retrieval on such a table possibly causes only one additional I/O rather than 100 additional I/Os.

For more information on overflow in hash tables, see Alternate Fill Factors (see page 216).

For ISAM tables, because the ISAM index is static, if you append a large number of rows, the table can begin to overflow. If there is no room on a page to append a row, an overflow page is attached to the data page. For example, if you wanted to insert empno #33, there is no more room on the data page, so an overflow page is allocated for the data page as shown in the following diagram:

```
Page 8                                  Overflow Page for Primary Page 8
|--------------------------- |        |-------------------------------|
| 29 |Ramos  | 31| 30000.000 |        |   33 |Quinn  | 33| 20000.000  |
| 30 |Brodie | 42| 40000.000 | --->   |                               |
| 31 |Smith  | 20| 10000.000 |        |                               |
| 32 |Horst  | 26| 50000.000 |        |                               |
|--------------------------- |        |-------------------------------|
```

For hash and ISAM tables, one way of looking at overflow is by looking at the tids of rows and analyzing the way the tids grow in a sequential scan through the table. For more information, see Tids (see page 202).

Example: Showing Overflow Distribution

The sample code shown here can be customized to show overflow distribution. Each time a primary page is encountered, the tid's value grows by 512. If a primary page has associated overflow pages, the tid's value jumps by more than 512. So if you run the embedded SQL/C program shown in Sample Code to Show Overflow, the output looks like that shown in Output from Sample Code.

Sample Code to Show Overflow

```
page_val = 0;
exec sql select key, tid
    into :key_val, :tid_val
    from tablename
exec sql begin;
  if (tid_val == page_val)
  {
    printf("Primary Page %d, tid = %d,",(page_val/512)+1, tid_val);
    printf(" Starting key value = %d0", key_val);
    page_val = page_val + 512;
    old_tid_val = tid_val;
    overflow_page = 0;
  }
  else
  {
    if (tid_val > old_tid_val + 1)
    {
      overflow_page++;
      printf("\n Overflow page %d,tid = %d0",over_page,tid_val);
    }
    old_tid_val = tid_val;
  }
exec sql end;
```

Output from Sample Code

```
Primary Page 1, tid = 0, Starting Key Value = 123
 Overflow page 1,tid = 2048
 Overflow page 2,tid = 2560
 Overflow page 3,tid = 3072
 Overflow page 4,tid = 3584
Primary Page 2, tid = 512, Starting Key Value = 456
 Overflow page 1,tid = 4096
 Overflow page 2,tid = 4608
 Overflow page 3,tid = 5120
 Overflow page 4,tid = 5632
```

B-tree Tables and Overflow

Eliminating overflow is one of the major benefits of the B-tree storage structure. Overflow in a B-tree occurs only at the leaf level, only when the page size is 2K, and only if you have a significant number of repetitive keys.

Note: The absence of overflow in a B-tree does not guarantee efficiency: it is still necessary to search all the rows for the specified repetitive key value across adjacent leaf pages.

For example, if 30 new employees all joined the company and all had the last name Aitken, the attempt is made to add their records to leaf page 1. In this case, because leaf page 1 can hold only 8 keys (remember that the leaf page can actually hold 2000/(*key_size* + 6)), an overflow leaf page is added to hold all the duplicate values. This is different than splitting the leaf page, because the same index pointer can still point to the same leaf page and be accurate. There are no additional key/leaf page entry added to the index.

In B-tree tables, you can look at overflow in the leaf level by running a query of the following type, substituting your B-tree table name for *t*, your B-tree keys for the *keycol* values, and the width of the key for *key_width*:

```
select keycol1, keycol2, overflow =
  (count(*)/keys_per_page)-1
  from tablename t
  group by keycol1, keycol2;
```

Notes:

- This query is not needed for a B-tree index, in which the automatic inclusion of the tidp column in the key prevents overflow.
- For B-tree tables with key compression selected, in the SELECT statement you can substitute an estimate of the average key size for *key_width*.
- For *keys_per_page* calculations, see the chapter “Calculating Disk Space.”

The results of this query give an approximation of the amount of overflow at the leaf level, per key value. The query works by calculating the number of keys that fit on a page and dividing the total number of particular key incidents—grouped by key—by this value. For instance, if there are 100 occurrences of a particular key and 10 keys fit on each page, there are nine overflow pages at the leaf level.

Other tables can incur overflow pages for reasons other than duplicate keys; hence, overflow distribution can involve more than simply running a query.

Secondary Indexes and Overflow

Overflow must be monitored in secondary indexes, as well as in the primary tables. Even if the base table has a low overflow percentage, the secondary indexes can badly overflow. Except when the base table is a heap or B-tree table, the base table generally overflows before the secondary index.

Secondary indexes need to be monitored and modified at interim points—even between base table modifications—to ensure a low percentage of overflow pages. More information is provided in Modifying Secondary Indexes (see page 226).

Chapter 11: Using the Query Optimizer

This section contains the following topics:

This chapter describes the query optimizer and how to use its features to obtain the best performance for your queries.

Data and Query Optimization

Ingres uses a query optimizer to develop sophisticated query execution strategies. The query optimizer makes use of basic information such as row size, number of rows, primary key fields and indexes defined, and more specific data-related information such as the amount of data duplication in a column.

The data-related information is available for use by the query optimizer only after statistics (see page 239) have been generated for the database. Without knowing exactly what data you have stored in your table, the query optimizer can only guess what your data looks like.

Consider the following examples:

```
select * from emp where empno = 13;
select * from emp where sex = 'M';
```

In each query, the guess is that few rows can qualify. In the first query, this guess is probably correct because employee numbers are usually unique. In the second query, however, this guess is probably incorrect because a company typically has as many males as females.

Why do restricted assumptions about your query make a performance difference? For a single-table, keyed retrieval where you are specifying the key, there is probably no difference at all. The key is used to retrieve your data. However, in a multi-table query with several restrictions, knowing what your data looks like can help determine the best way to execute your query. The following example shows why:

```
select e.name, e.dept, b.address
  from emp e, dept d, bldg b
  where e.dept = d.dname
  and d.bldg = b.bldg
  and b.state = 'CA'
  and e.salary = 50000;
```

There are many ways of executing this query. If appropriate keys exist, the probable choice is to execute the query in one of these two ways:

- Retrieve all the employees with a salary of 50000. Join the employees with a salary of 50000 to the department table, join the employees with their valid departments to the valid buildings. The tables are processed in the following order:

  emp --> dept --> bldg

- Retrieve all the buildings with a state of CA. Join the valid buildings with the department table, and join the qualifying departments to the valid employees. The tables are processed in the following order:

  bldg --> dept --> emp

The difference between these two possibilities is the order in which the tables are joined. Which method is preferable? Only if you knew exactly how many employees made $50,000, how many buildings were in California, and how many departments were in each building, can you pick the best strategy.

The best (that is, the fastest) query execution strategy can be determined only by having an idea of what your data looks like—how many rows qualify from the restriction, and how many rows join from table to table.

Query Execution Plans (QEPs) (see page 258), generated by the query optimizer each time you perform a query, illustrate how a query is executed. By optimizing your database, you can optimize the QEPs that are generated, thereby making your queries more efficient.

Database Statistics

When you generate statistics for a database, you are optimizing the database, which affects the speed of query processing. More complete and accurate statistics generally result in more efficient query execution strategies, which further result in faster system performance.

The extent of the statistics to be generated for a database can be modified by various options, including restricting the tables and columns that are used.

Note: Deleting all the rows in a table does not delete the statistics.

Generate Statistics

You can generate database statistics by issuing the optimizedb command at the command line. For more information, see the *Command Reference Guide*.

In VDBA, select a database, right-click, and choose Generate Statistics. On the Optimize Database dialog, select the desired options. For more information, see the VDBA online help.

Assumptions of the Query Optimizer

If a database has not been optimized, the query optimizer assumes that:

- All exact match restrictions return 1% of the table, except where a key or index is defined to be unique, in which case one row is returned for the indexed attribute:

  where emp.empno = 275

  Note: To override the default of 1% for exact match qualifications, use the Configuration-By-Forms opf_exact_key parameter.

- All range qualifications (<, <=, >=, >) and like predicates, in which the first character is **not** a wild card, return 10% of the table for each qualification. Thus, if there are three (non-exact match) qualifications, the following amount of the table is selected:

  1/10 x 1/10 x 1/10 = 1/1000

  Note: To override the default of 10% for range qualifications, use the CBF opf_range_key parameter.

- All "not equals" qualifications (<>) and like predicates, in which the first character is a wild card, return 50% of the table for each qualification. The default 50% for these qualifications can be overidden by the Configuration-By-Forms opf_non_key parameter.

  All joins are assumed to be one-to-one, based on the smaller data set; for example, when table1 with 100 rows is joined with table2 with 1000 rows, the estimated result is 100 rows.

- When there are restrictions on the join tables, the number of resulting rows is greater than or equal to the lower bound of 10% of qualifying rows from the smaller table.

If these assumptions are not valid for your data, you must optimize the database by generating statistics for it.

Resources Required During Optimization

Optimizing a database generally requires disk space, because temporary tables are created.

While the optimization process is running, the locking system takes an exclusive lock for a brief period on the user table being optimized. Whenever possible, tables on which statistics are created are not locked while statistics are gathered. This means that updates to the optimized table are not disabled for long periods. However, it is recommended that you optimize the database during off-hours.

When running optimizedb from the command line, the "-o filename" option can be used to write the statistics to an external file rather than to the system catalogs, so as to not require any catalog locks. At a later, more convenient time, the statistics can be loaded into the catalog with the "-i filename" option, using the same external file. Optimizedb -o requires read locks and optimizedb -i requires a brief exclusive lock on the user table. For more information on the optimizedb command, see the *Command Reference Guide*.

System Modification After Optimization

Because optimizing a database adds column statistics and histogram information to the system catalogs, you should run the system modification operation on your database after optimizing it.

Running system modification modifies the system tables in the database to optimize catalog access. You should do this on a database periodically to maintain peak performance.

Run System Modification

To run system modification, use either of the following methods.

At the command line, use the sysmod command. For more information, see the *Command Reference Guide*.

In VDBA, use the System Modification dialog, as described in online help topic Optimizing System Tables. For a complete description of all the options, see online help for the System Modification dialog.

Example: Run System Modification in VDBA

To specify only those system tables affected by the optimization process, do the following:

1. Open the System Modification dialog in VDBA.
2. Enable Specify in the Tables group box.
3. Enable iihistogram and iistatistics in the resulting list box, and click OK.

Information Collected by the Optimizer

When you optimize a database, the following information is collected:

- The number of unique values in those columns selected for optimization
- A count showing what percentage of those column values are NULL
- The number of duplicate values there are in those columns in the whole table, on average, or whether all values are unique. This is termed the repetition factor.
- A histogram showing data distribution for each of those columns. Sample histograms and further information on their contents can be found in Optimization Output (see page 249).

The query optimizer can use this information to calculate the cost of a particular QEP.

Types of Statistics to Generate

When optimizing a database, you can create several types and levels of statistics by specifying options.

First, you can specify what set of data the statistics are gathered on:

- Non-sampled statistics—all rows in the selected tables are retrieved
- Sampled statistics—a subset of rows from the selected tables is retrieved

Next, either of the following can be created, based on the selected data set. This division determines how much information about the distribution of data the statistics can hold:

- Full statistics—a histogram for the whole range of column values is created
- Minmax statistics—a histogram showing only minimum and maximum values is created

Non-Sampled and Sampled Statistics

When generating statistics for a database, by default all rows of the selected tables are used in the generation of statistics. These non-sampled statistics represent the most accurate statistics possible, because all data is considered.

When the base table is large, you must use sampled statistics. With a sufficient sampling, statistics created are almost identical to statistics created on the full table. The processing for sampled statistics is discussed in greater detail in Sampled Optimizer Statistics (see page 297).

Note: By default, optimizedb uses sampled statistics for tables that have more than 500,000 rows.

Generate Sampled Statistics

In VDBA, to specify a percentage of rows to be sampled, enable Statistics on Sample Data check box, and specify the percentage using the Percentage control in the Optimize Database dialog.

Full Statistics

When optimizing a database, full statistics are generated by default. Full statistics carry the most information about data distribution (unless the data is modified significantly after statistics are collected).

The cost of their creation (in terms of system resources used), however, is the highest of all types. For each selected column the table is scanned once, and the column values are retrieved in a sorted order. Depending on the availability of indexes on the selected columns, a sort can be required, increasing the cost even further.

The process of generating such complete and accurate statistics can require some time, but there are several ways to adjust this.

Generate Full Statistics on Sample Data

You can shorten the process of creating full statistics by enabling the Statistics on Sample Data check box in VDBA.

This example generates full statistics with a sampling of 1% rows of the emp table:

1. In VDBA, open the Optimize Database dialog for the database. (For more information, see online help.) Specify the following:
   - Enable the Statistics on Sample Data check box.
   - Enter **1** for the Percentage.
   - Enable the Specify Tables check box, and then click Tables.

   The Specify Tables dialog appears.
2. Enable the emp table, and click OK.

   You are returned to the Optimize Database dialog.
3. Click OK.

Minmax Statistics

Minmax statistics are “cheaper” than full statistics to create. In most cases they require only one scan of the entire table. Statistics created have information only about minimum and maximum values for a column. This can be acceptable if the distribution of values in the column is reasonably even. However, if the values of a particular column are skewed, minmax statistics can mislead the query optimizer and result in poor query plan choices.

In VDBA, to specify minmax statistics, enable the Min Max Values check box in the Optimize Database dialog.

Example: Generate Statistics with Only Minimum and Maximum Values for a Table

This example generates statistics with only minimum and maximum values for the employee table:

1. In VDBA, open the Optimize Database dialog for the table. For more information, see online help.
2. Enable the Min Max Values check box.
3. Enable the Specify Tables check box.
4. Click Tables to open the Specify Tables dialog.
5. Enable the employee table, and click OK.
6. Click OK.

Key Column Statistics

Key column statistics create full or minmax statistics on key or indexed columns only. These statistics are generated by enabling the Gen Statistics on Keys/Index check box in the Optimize Database dialog. The effect of this option is the same as specifying key and index columns for a table using the Specify Columns dialog. For more information, see Generating Database Statistics in online help. Using the Gen Statistics on Keys/Index check box saves you some work by determining from the catalogs which columns are keys and indexed.

Examples: Create Statistics on Key or Indexed Columns Only

This example generates full statistics for all key and indexed columns in the employee table:

1. In VDBA, open Optimize Database dialog for the table. For more information, see online help.
2. Enable the Gen Statistics on Keys/Index check box.
3. Enable the Specify Tables check box.
4. Click Tables to open the Specify Tables dialog.
5. Enable the employee table, and click OK.
6. Click OK.

To generate minmax statistics on a 1% sampling, do the following in the Optimize Database dialog:

1. Enable the Gen Statistics on Keys/Index check box.
2. Enable the Min Max Values check box.
3. Enable the Statistics on Sample Data check box.
4. Enter **1** for the Percentage.
5. Enable the Specify Tables check box.
6. Click Tables to open the Specify Tables dialog.
7. Enable the employee table, and click OK.
8. Click OK.

All key and indexed columns in the table are processed regardless of any column designations specified using the Specify Columns dialog. For example, assume that dno is a data column and kno is a key column in the employee table.

The following example for generating full statistics is the same as the first example in this section, except that in addition to key and index columns, statistics are generated also for the dno column:

1. Enable the Gen Statistics on Keys/Index check box.
2. Enable the Specify Tables check box.
3. Click Tables to open the Specify Tables dialog.
4. Enable the employee table, and click OK
5. Enable the Specify Columns check box.
6. Click Columns to open the Specify Columns dialog.
7. Enable the dno and kno columns, and click OK.
8. Click OK.

The kno column designation in Step 6 is superfluous, because this is a key column and the Gen Statistics on Keys/Index check box is enabled.

Statistics from an Input Text File

Statistics can be read in from a text file. The input file must conform to a certain format, which is identical to that produced when you direct output to a file when displaying statistics. Display Optimizer Statistics (see page 292) provides more information.

The file can be edited to reflect changes in data distribution as required, before submitting the file for use during the optimization process. However, this can potentially mislead the query optimizer into generating poor query plans. Manually editing statistics must be done only if you have a full understanding of the data and how the statistics are used in Ingres.

Details on creating and using text files as input when optimizing a database are provided in Statistics in Text Files (see page 294).

Column Statistics

Collecting statistics is generally a time-consuming process because a large amount of data must be scanned. The techniques described so far—except for Key Column Statistics (see page 245)—collect statistics on all columns of the indicated tables.

It is not necessary, however, to choose all columns in all tables in your database when optimizing. The query optimizer uses statistics on a column only if the column is needed to restrict data or if it is specified in a join. **Therefore, it is a good idea to limit creation of statistics only to those columns used in a WHERE clause.**

The DBA or table owner usually understands the table structure and content and is able to predict how the various columns are used in queries. Thus, someone familiar with the table can identify columns that are used in the WHERE clause.

Given these queries:

```
select name, age from emp
  where dept = 'Tech Support';

select e.name, e.salary, b.address
  from emp e, bldg b, dept d
  where e.dept = d.dname
  and d.bldg = b.bldg;
```

Candidate columns for optimization are:

emp table:	dept
dept table:	dname, bldg
bldg table:	bldg

Based on their use in these sample queries, there is no reason to obtain statistics on employee name, age, salary, or building address. These columns are listed in the target list only, not the WHERE clause of the query.

Columns used in the WHERE clause are often indexed to speed up joins and execution of constraints. If this is the case, specify the Gen Statistics on Keys/Index option to create statistics on key (that is, indexed) columns. However, it is often just as important to create statistics on non-indexed columns referenced in WHERE clauses.

Create Statistics on Keys

In VDBA, to create statistics on key (that is, indexed) columns, enable the Gen Statistics on Keys/Index check box in the Optimize Database dialog.

Histogram (Optimization Output)

When you optimize a database, output is generated to show the statistics.

For example, if the Print Histogram option was enabled when optimizing the database, and you chose to optimize the name and sex columns of the emp table, the following output is typical:

```
*** statistics for database demodb version: 00850
*** table emp1 rows:1536 pages:50 overflow pages:49
*** column name of type varchar (length:30, scale:0, nullable)
date:2000_02_24 15:40:38 GMT  unique values:16.000
repetition factor:96.000 unique flag:N complete flag:0
domain:0 histogram cells:32 null count:0.0000       value length:8
cell:   0    count:0.0000    repf:0.0000    value:Abbot  \037
cell:   1    count:0.0625    repf:96.0000    value:Abbot
cell:   2    count:0.0000    repf:0.0000    value:Beirne \037
cell:   3    count:0.0625    repf:96.0000    value:Beirne
cell:   4    count:0.0000    repf:0.0000    value:Buchanam
cell:   5    count:0.0625    repf:96.0000    value:Buchanan
cell:   6    count:0.0000    repf:0.0000    value:Cooper \037
cell:   7    count:0.0625    repf:96.0000    value:Cooper
cell:   8    count:0.0000    repf:0.0000    value:Dunham \037
cell:   9    count:0.0625    repf:96.0000    value:Dunham
cell:  10    count:0.0000    repf:0.0000    value:Ganley \037
cell:  11    count:0.0625    repf:96.0000    value:Ganley
cell:  12    count:0.0000    repf:0.0000    value:Hegner \037
cell:  13    count:0.0625    repf:96.0000    value:Hegner
cell:  14    count:0.0000    repf:0.0000    value:Jackson\037
cell:  15    count:0.0625    repf:96.0000    value:Jackson
cell:  16    count:0.0000    repf:0.0000    value:Klietz \037
cell:  17    count:0.0625    repf:96.0000    value:Klietz
cell:  18    count:0.0000    repf:0.0000    value:Myers  \037
cell:  19    count:0.0625    repf:96.0000    value:Myers
cell:  20    count:0.0000    repf:0.0000    value:Petersom
cell:  21    count:0.0625    repf:96.0000    value:Peterson
cell:  22    count:0.0000    repf:0.0000    value:Rumpel \037
cell:  23    count:0.0625    repf:96.0000    value:Rumpel
cell:  24    count:0.0000    repf:0.0000    value:Singer \037
cell:  25    count:0.0625    repf:96.0000    value:Singer
cell:  26    count:0.0000    repf:0.0000    value:Stec   \037
cell:  27    count:0.0625    repf:96.0000    value:Stec
cell:  28    count:0.0000    repf:0.0000    value:Washings
cell:  29    count:0.0625    repf:96.0000    value:Washingt
cell:  30    count:0.0000    repf:0.0000    value:Zywicki\037
cell:  31    count:0.0625    repf:96.0000    value:Zywicki
unique chars: 14 9 11 11 9 11 6 3
char set densities: 0.5200 0.3333 0.4762 0.6667 0.0952 0.1111 0.0633 0.0238

*** statistics for database demodb version: 00850
 *** table emp rows:1536 pages:50 overflow pages:49
 *** column sex of type char (length:1, scale:0, nullable)
 date:23-feb-2000 10:12:00     unique values:2.000
 repetition factor:768.000 unique flag:N complete flag:0
 domain:0 histogram cells:4 null count:0.0000000       value length:1
 cell:   0    count:0.0000000         repf:0.0000000         value:E
 cell:   1    count:0.0006510         repf:1.0000000         value:F
 cell:   2    count:0.0000000         repf:0.0000000         value:L
 cell:   3    count:0.9993489         repf:1535.0000000      value:M
 unique chars: 2
char set densities: 0.1428571
```

The items in the histogram are as follows:

database

Database name

version

Version of the catalog from which statistics were derived. Shown only if version is 00605 or later.

table

Table currently processing

rows

Current number of rows in table as stored in the iitables catalog

pages

Number of pages (from the iitables catalog)

overflow pages

Number of overflow pages (from the iitables catalog)

column

Column currently processing

type

Column data type. The length, scale, and nullable indicators are obtained from the iicolumns catalog.

date

Time and date when statistics were created

unique values

Number of unique values found in the table

repetition factor

Average number of rows per unique value. The repetition factor times the number of unique values must produce the row count.

unique flag

"Y" if unique or nearly unique, "N" if not unique

complete flag

All possible values for the column exist in this table. When this column is used in a join predicate with some other column, it tells the query optimizer that every value in the other column must be a value of this column as well. This knowledge enables the query optimizer to build more accurate query plans for the join.

domain

Not used

histogram cells

Number of histogram cells used (0 to 500 maximum)

null count

Proportion of column values that are NULL, expressed as a real number between 0.0 and 1.0

value length

Length of cell values

cell

For each cell, a cell number, count (proportion of rows whose values fall into this cell: between 0.0 and 1.0), average number of duplicates per unique value in the cell, and the upper bound value for the cell

unique chars

Number of unique characters per character position. Shown only for character columns.

char set densities

Relative density of the character set for each character position. Shown only for character columns.

The number of unique values the column has is calculated. The count listed for each cell is the fraction of all the values falling between the lower and upper boundaries of the cell. Statistics for the sex column show that there are no rows with values less than or equal to 'E,' 0.06510% of rows with values equal to 'F,' no rows with values in the 'G' to 'L' range, and 99.93% of the rows with values equal to 'M.' The cell count includes those rows whose column values are greater than the lower cell bound but less than or equal to the upper cell bound. All cell counts must add to 1.0, representing 100% of the table rows.

Looking at the cells for the name column, you see that between the lower bound cell 0, “Abbot \037”, and cell 1, “Abbot”, 6.25% of the employee’s names are located:

```
cell:   0   count:0.0000000   repf:0.0000000  value:Abbot   \037
cell:   1   count:0.0625000   repf:96.000000  value:Abbot
```

A restriction such as the following brings back about 6.25% of the rows in the table:

```
where emp.name = 'Abbot'
```

The character cell value \037 at the end of the string is octal for the ASCII character that is one less than the blank. Therefore, cell 0 in the name example represents the value immediately preceding ‘Abbot’ in cell 1. This indicates that the count for cell 1 includes all rows whose name column is exactly ‘Abbot.’

In addition to the count and value, each cell of a histogram also contains a repetition factor (labeled “repf” in the statistics output). This is the average number of rows per unique value for each cell, or the “per-cell” repetition factor. The query optimizer uses these values to compute more accurate estimates of the number of rows that result from a join. This is distinct from the repetition factor for the whole column displayed in the header portion of the statistics output.

Histogram Cells

A histogram can have up to 15,000 cells. The first cell of a histogram is always an “empty” cell, with count = 0.0. It serves as the lower boundary for all values in the histogram. Thus, all values in the column are greater than the value in the first cell. This first cell is usually not included when discussing number of cells, but it is included when statistics are displayed.

A histogram in which there is a separate cell for each distinct column value is known as an “exact” histogram. If there are more distinct values in the column than cells in the histogram, some sets of contiguous values must be merged into a single cell. Histograms in which some cells represent multiple column values are known as “inexact” histograms.

You can control the number of cells used, even for inexact histograms. You can choose to set the number of inexact cells to the same number you chose for an exact histogram, or to another number that seems appropriate. If your data is unevenly distributed, the data distribution cannot be apparent when merged into an inexact histogram with the default 100 cells. Increasing the number of cells can help.

You can control the number of cells your data is merged into even if you go above the maximum number of histogram cells you requested. You can choose to set the default merging number to the same number you chose for the maximum, or a lesser number, if the default of 100 cells seems inappropriate. If your data is unevenly distributed, the data distribution cannot be apparent when merged into the default 100 cells, and controlling the merging factor can help.

To control the maximum histogram cells, use the Max Cells “Exact” Histogram option in the Optimize Database dialog (the maximum value accepted is 14,999). You can control the number of cells that your data is merged into if you go beyond the maximum number of unique values using the Max Cells “Inexact” Histogram option in the Optimize Database dialog. By default, the number of cells used when merging into an inexact histogram is 100, and the maximum value is 14,999.

For example, set the maximum number of unique histogram cells to 200, and if there are more than 200 unique values, merge the histogram into 200 cells. To do this, set both the Max Cells “Exact” Histogram and the Max Cells “Inexact” Histogram options in the Optimize Database dialog to 200.

Set the maximum number of unique histogram cells to 100, and if there are more than 100 unique values, merge the histogram into 50 cells. To do this, set Max Cells “Exact” Histogram to 100 and Max Cells “Inexact” Histogram to 50.

When using these options, remember that the goal is to accurately reflect the distribution of your data so that there can be an accurate estimate of the resultant number of rows from queries that restrict on these columns. The query optimizer uses linear interpolation techniques to compute row estimates from an inexact histogram and the more cells it has to work with, the more accurate are the resulting estimates. The cost of building a histogram is not dependent on the number of cells it contains and is not a factor when determining how many cells to request.

Statistics and Global Temporary Tables

Because global temporary tables only exist for the duration of an Ingres session, Optimize Database cannot be used to gather statistical information about them. Without histograms, the query optimizer has no knowledge about the value distributions of the columns in a global temporary table. Ingres maintains a reasonably accurate row count for global temporary tables, and this row count can be used by the query optimizer to compile a query which accesses a global temporary table.

The row counts alone are usually enough to permit the compilation of efficient query plans from queries that reference global temporary tables, in particular because they often contain relatively small data volumes. The lack of histograms on global temporary tables, however, can cause poor estimates of the number of rows that result from the application of restriction or join predicates. These poor estimates can in turn cause the generation of inefficient query plans. Inefficient query plans typically occur with large global temporary tables or tables with columns having skewed value distributions, which are not handled well by the default estimation algorithms of the query optimizer.

To help deal with such situations, there is a mechanism available to associate “model” histograms with global temporary tables.

How to Associate "Model" Histograms with Global Temporary Tables

Associating "model" histograms with global temporary tables can help alleviate the generation of inefficient query plans that can typically occur with large global temporary tables or tables with columns having skewed value distributions.

To associate "model" histograms with global temporary tables, follow these steps:

1. Create a persistent table with the same name as the global temporary table being modeled. The schema qualifier for the table must be either the user ID of the executing user of the application creating and accessing the global temporary table, or the special user ID "_gtt_model". Its column definitions must include at least those from the global temporary table for which histograms are to be built. The column names and types must exactly match those of the global temporary table.
2. Populate the persistent table with a set of rows, which is representative of a typical instance of the global temporary table.
3. Run optimizedb on those columns of the persistent table for which histograms are desired (typically, the columns contained in WHERE clauses in any referencing queries).
4. After the histograms have been built, the persistent table can be emptied of rows, to release the space it occupies. This must be done with a DELETE FROM xxx statement, to delete the rows but leave the catalog definition (and histograms).

When the query optimizer analyzes WHERE clause predicates with columns from a global temporary table, it looks for the catalog definition of a similarly named persistent table with a schema qualifier matching the ID of the executing user or _gtt_model. If one is found, it looks for histograms on similarly named columns whose type and length exactly match those of the global temporary table columns. If these conditions are satisfied, it uses the model histograms.

Not all faulty query plans involving global temporary tables can be improved this way. The modeling technique depends on the fact that all or most instances of the global temporary table have similar value distributions in the histogrammed columns. If this is not true, a single instance of the table (as with the model persistent table) will not be representative of them all, and can improve the query plans in some executions of the application, but degrade other executions.

When to Rerun Optimization

Optimization does not necessarily need to be run whenever data is changed or added to the database. Optimization collects statistics that represent percentages of data in ranges and repetition factors. For instance, the statistics collected on employee gender show that 49% of the employees are female and 51% are male. Unless this percentage shifts dramatically, there is no need to rerun optimization on this column, even if the total number of employees changes.

You must rerun optimization if there are modifications to the database that alter the following:

- Repetition factor
- Percentage of rows returned from a range qualification (that is, your histogram information is incorrect)

For example, if you had run complete statistics on the empno column early in your company's history, your repetition factor is correct because all employees still have unique employee numbers. If you used ranges of employee numbers in any way, as you added new employees your histogram information is less accurate.

If your company originally had 100 employees, 10% of the employees have employee numbers greater than 90. If the company hired an additional 100 employees, 55% of the employees have employee numbers greater than 90, but the original histogram information does not reflect this.

Columns that show this type of "receding end" growth and are used in range queries can periodically need to have optimization run on them (exact match on employee number is not affected, because the information that says all employee numbers are unique is still correct).

Even if the statistics are not up-to-date, the query results are still correct.

Example: Before and After Optimization

If statistics are available on a column referenced in a WHERE clause, the query optimizer uses the information to choose the most efficient QEP. Understanding how this information is used can be helpful in analyzing query performance. For more information, see Query Execution Plans (see page 258).

Two QEPs showing the effect of optimization are presented here. The first is a QEP before optimizing; the second shows the same query after optimization. The query used is a join, where both the r and s tables use the B-tree storage structure:

```
select * from r, s
  where s.a > 4000 and r.a = s.a;
```

QEP Before Optimization

Before obtaining statistics, the optimizer chooses a full sort-merge (FSM) join, because it assumes that 10% of each table satisfies the qualification “a > 4000,” as shown in the QEP diagram below:

```
QUERY PLAN 4,1, no timeout, of main query
                 FSM Join(a)
                 Heap
                 Pages 1 Tups 267
                 D15 C44
               /              \
          Proj-rest           Proj-rest
          Sorted(a)           Sorted(a)
          Pages 5 Tups 1235   Pages 1 Tups 267
          D11 C12             D4 C3
            /              /
   r                       s
   B-Tree(a)               B-Tree (a)
   Pages 172 Tups 12352    Pages 37 Tups 2666
```

QEP After Optimization

After obtaining statistics, the optimizer chooses a Key join, because only one row satisfies the qualification “a > 4000,” as shown in the QEP diagram below:

```
QUERY PLAN 4,1, no timeout, of main query
                 K Join(a)
                 Heap
                 Pages 1 Tups 1
                 D4 C1
               /              \
         Proj-rest          r
         Sorted(a)          B-Tree(a)
         Pages 1 Tups 1     Pages 172 Tups 12352
         D2 C1
           /
  s
  B-Tree(a)
  Pages 37 Tups 2666
```

The cost of the key join is significantly less than the cost of the FSM join because the join involves far fewer rows.

Query Execution Plans

When the query optimizer evaluates a query (such as the SQL statements SELECT, INSERT, UPDATE, DELETE, and CREATE TABLE...AS), it generates a QEP showing how the query is executed. Once the QEP has been generated, it can be used one or more times to execute the same query. Because there are often many different ways to optimize a given query, choosing the best QEP can have a significant impact on performance.

You can display a diagram or graph of the QEP selected, which can be used to gain insight into how queries are handled by the query optimizer. Knowing how to read and analyze QEPs can allow you to detect, and often avoid, hidden performance problems. After examining a QEP you can, for example, decide that you need to optimize your database to provide the optimizer with better statistics, as described in Database Statistics (see page 239).

Note: Examining QEPs can help you understand what is involved in executing complex queries in single-user situations. For multi-user performance issues, see the chapter “Understanding the Locking System.”

Information on a QEP

The information that can appear on a QEP is as follows:

Table or Index Name

Indicates the table on which the query is being run or the secondary index, if any is selected by the query optimizer for execution of the query. This information is provided for orig nodes only (described under Type of Nodes in a QEP below).

Label

Indicates the type of node. For example, Proj-rest identifies a projection-restriction node (described under Type of Nodes in a QEP below).

Storage Structure

Indicates the storage structure in use, as follows, where *key* is the primary key, and NU indicates that the key structure cannot be used:

B-tree(*key*|NU)

Hashed(*key*|NU)

Heap

Isam(*key*|NU)

Total Number of Pages and Tuples

Indicates the total number of pages involved at the node, and the total number of tuples (rows).

Query Cost Information

Indicates the cumulative amounts of cost that are anticipated at each stage in the execution of the query. This cost is a blend of the CPU consumption and the number of disk I/Os involved in plan execution. The information is shown in the following form:

- Dx estimates the disk I/O cost. x approximates the number of disk reads to be issued.
- Cy estimates the CPU usage, which has been subjected to a formula which turns it into an equivalent number of disk I/Os. y units can be used to compare amounts of CPU resources required.
- Nz is shown for Star databases only. z represents the network cost of processing the query.

Because these values are cumulative up the tree, the top node carries the total resources required to execute the query. The cost involved in executing a specific node is, therefore, the values for that node, minus those of the child node (or both child nodes in the case of a join node).

The QEP graph you see in VDBA indicates both the cumulative cost and the cost for the individual node. For more information, see Viewing QEP Node Information in online help.

View a QEP

In general, it is a good idea to test run a query (that is, view the QEP).

In VDBA, if you open the SQL Scratchpad window and click Execute QEP, you automatically see the query execution plan in a graphical form. For step-by-step instructions, see Viewing the Query Execution Plan in online help.

From a terminal monitor or embedded SQL, you can see the QEP by using the SET QEP statement. On the SET statement, the [NO]OPTIMIZEONLY, [NO]QEP, and JOINOP [NO]TIMEOUT clauses pertain to QEPs. For more information, see the *SQL Reference Guide*. The QEP is displayed in text-only format when using SQL.

Control QEP Generation Using a Environment Variable

To control whether QEPs are generated using an operating system environment variable, issue the following commands:

Windows:

```
set ING_SET=set qep
```

UNIX:

C shell:

```
setenv ING_SET "set qep"
```

Bourne shell:

```
ING_SET = "set qep"
```

```
export ING_SET
```

VMS:

```
define ING_SET "set qep"
```

Text-Only QEP

In a terminal monitor, a QEP is displayed as a tree, where each node has one or two children:

```
               Parent
              /     \
          Child    Child
          /
      Parent
      /     \
 Child    Child
```

Only join nodes have two children. Other nodes have a left child only. Information on node types is provided in Types of Nodes in a QEP (see page 263).

The tree is read from bottom to top and from left to right, starting with the child nodes at the bottom, going up to the intermediate child nodes, if any, and finally up to the first parent node:

QEPs as Data Flow Trees

The bottom up approach in the tree diagram mirrors the flow of data during the execution of a query plan.

Rows are read in the leaf nodes of the query plan, WHERE clauses are applied to reduce the number of rows as soon as possible, with qualified rows being passed up through the remaining nodes of the query plan.

Intermediate plan nodes can sort the data or join it to rows from other tables.

Each successive node performs some refinement on the rows received from below. The final result rows emerge from the top of the plan as requested by the query.

Modes for Showing Tree Diagrams

In the SQL Scratchpad window, you can show the tree diagram in one of two modes:

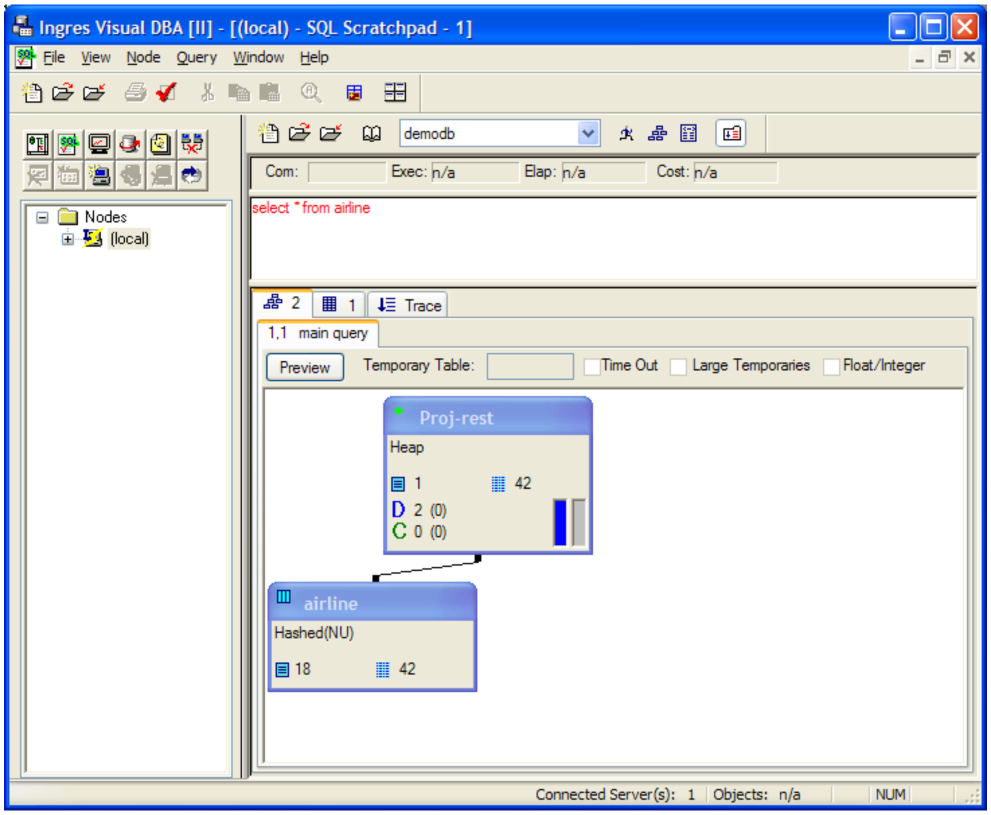
- Preview mode gives you a condensed version of the tree, where you can point to a particular node to see its detailed information.
- Normal mode displays the detailed information as part of the tree diagram.

Note: A query that has been modified to include views, integrities, and/or grants, is more involved. The QEP diagram for an involved query (as shown by set qep) can be very wide. On a printout, the diagram can even wrap so that similar levels in the tree appear on different levels. You can find it easier to read such a QEP if you cut and paste the diagram into the correct levels.

Graphical QEP

In VDBA, QEP diagrams appear in the query information pane as a graph.

For a detailed description of each element in the graph and the meaning of the colors, see Viewing QEP Node Information in online help.



```
select * from airline
```

Types of Nodes in a QEP

Each node in a QEP has detailed information associated with it, depending on the type of node.

The types of nodes are as follows:

- Orig (or leaf) node—describes a particular table
- Proj-rest node—describes the result of a projection and/or WHERE clause restriction applied to a subsidiary node
- Join node—describes a join. One of the following strategies is used:
  - Cartesian product
  - Full sort merge
  - Partial sort merge
  - Key and tid lookup joins
  - Subquery joins
- Exchange node—describes a point at which separate plan fragments execute concurrently on different threads as part of a parallel query plan

Sort Nodes in a QEP

Many types of nodes can also be shown as *sort* nodes. A sorting node causes the data to be sorted as it is returned. Any node other than an orig node can appear with a sort indication. A query with a SORT clause has a sort node as the topmost node in the QEP. This type of node displays:

- Total number of pages and tuples
- Query cost information

For a description of each of these parts of the display, see Information on a QEP (see page 259).

Sort nodes make use of a sort buffer and so consume primarily CPU resources, requiring disk I/O only when the volume of data to be sorted cannot be accommodated in the sort buffer. The heapsort algorithm is used; it is very fast on both unordered data and data which is nearly sorted.

Non-Join Nodes in a QEP

Types of non-join nodes are as follows:

- Orig
- Projection-restriction
- Exchange

Orig Nodes

Orig nodes are nodes with no children. When reading a QEP, you should first find the orig nodes of the tree. Orig nodes are the most common type of QEP node.

Orig nodes describe a base table or secondary index being accessed from the query. This type of node displays the following:

- Table or index name
- Storage structure
- Total number of pages and tuples

For a description of each of these parts of the display, see Information on a QEP (see page 259).

Projection-Restriction Nodes

A projection-restriction (proj-rest) node describes the result of a projection and/or WHERE clause restriction applied to a subsidiary node. It defines how a subsidiary node is to key into the table and what qualification to use on the table. This type of node displays the following:

- A label identifying it as a proj-rest node
- Storage structure (which is usually heap)
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

For a description of each of these parts of the display, see Information on a QEP (see page 259).

Proj-rest nodes are used to remove data irrelevant to the query from a table, so that the minimum amount of data is carried around during the execution of the query. Therefore, only those columns referenced in the target list and WHERE clause for a table are projected, and only those rows that satisfy the WHERE clause restrictions are retained for use higher in the plan.

All you see is the amount of disk I/O required to read the appropriate rows from the node below, and that amount depends on what storage structures were used and the number of pages accessed.

Exchange Nodes

Exchange nodes appear in parallel query plans. An exchange node results in one or more threads being spawned to execute the plan fragment beneath the node. It allows different portions of a complex query plan to execute concurrently, reducing the overall elapsed time taken to execute the query. This type of node displays:

- Estimated reduction in execution time due to the presence of the exchange node
- Count of child threads spawned to execute the plan fragment below the exchange node
- PC Join count, the number of join fragments performed by a partition compatible join

For more information, see Parallel Query Execution (see page 283).

Examples of Non-join Nodes

Here are QEP examples that illustrate non-join nodes. The Sample Tables section describes the tables and indexes used in these examples.

Sample Tables

In these examples, the following two tables are used:

1. Table arel(col1, col2, col3):

```
Name:                        arel
Owner:                       supp60
Created:                     26-oct-1998 08:50:00
Location:                    ii_database
Type:                        user table
Version:                     II2.5
Row width:                   413
Number of rows:              156
Storage structure:           hash
Duplicate Rows:              allowed
Number of pages:             70
Overflow data pages:         6
```

Column Information:

```
Column                                       Key
Name   Type      Length   Nulls   Defaults  Seq
col1   integer    4       yes       no       1
col2   integer    4       yes       no
col3   varchar    400     yes       no
Secondary indexes: aindex (col2)  structure: isam
```

2. Table brel(col1,col2):

```
Name:                        brel
Owner:                       supp60
Created:                     26-oct-1998 08:53:00
Location:                    ii_database
Type:                        user table
Version:                     II2.5
Row width:                   10
Number of rows:              156
Storage structure:           isam
Duplicate Rows:              allowed
Number of pages:             3
Overflow data pages:         1
```

Column Information:

```
Column                                         Key
Name    Type      Length   Nulls   Defaults   Seq
col1    integer    4       yes       n         1
col2    integer    4       yes       no
Secondary indexes: none
```

Primary Key Lookup

This is an example of a simple primary key lookup. The QEP is shown below for the following SQL query:

```
select col1, col2 from arel
  where col1 = 1234
  order by col2;

QUERY PLAN 3,1, no timeout, of main query
                Sort Keep dups
                Pages 1 Tups 1
                D1 C0
                /
            Proj-rest
            Sorted(col1)
            Pages 1 Tups 1
            D1 C0
            /
         arel
         Hashed(col1)
         Pages 70 Tups 156
```

Reading this QEP diagram from bottom to top, Hashed(col1) means the row is being read through the index to return only those rows for which "col1 = 1234," as opposed to Hashed(NU) where NU indicates that the key structure cannot be used and all rows are returned. The projection-restriction node selected the rows matching the where constraint and removed superfluous columns. The final sort node reflects the SORT clause on the query.

Select on a Non-Keyed Field

The following is an example of a select on a non-keyed field. The QEP is shown below for the following SQL query:

```
select col1, col2 from arel
  where col3 = 'x'
  order by col1;

QUERY PLAN 3,1, no timeout, of main query
                    Sort Keep dups
                    Pages 1 Tups 1
                    D9 C0
                    /
                 Proj-rest
                 Heap
                 Pages 1 Tups 1
                 D9 C0
                  /
              arel
              Hashed(NU)
              Pages 70 Tups 156
```

In this example the Hashed(NU) implies that the table had to be scanned (that is, all 70 pages had to be visited). Without optimization statistics, the query optimizer uses a best guess approach (1% for equalities and 10% for non-equalities).

The query optimizer takes into account disk read-ahead or group reads when performing scans of tables—although 70 pages of data have to be read to scan arel, the estimated disk I/O value is only nine reads (D9) due to this effect. The query optimizer assumes a typical read-ahead of eight pages when performing scans, so here 70/8 reads generates an estimate of nine disk operations.

Join Nodes in a QEP

There is an inner and an outer tree beneath every join node, which function similarly to an inner and outer program loop. By convention, the left-hand tree is called the outer tree, and the right-hand tree is called the inner tree.

There are various types of join nodes, described individually below, but the joining method is the same: for each row from the outer tree, there is a join to all of the rows that can possibly qualify from the inner tree. The next row from the outer tree is processed, and so on.

Any join node can have outer join information if an outer join is present.

Cartesian Product Node

The Cartesian product, or *cart-prod*, strictly follows the un-optimized join model, with each row in the outer node compared to all rows from the inner node. This does not mean that all rows are actually read, only that all rows that satisfy the conditions of the query are compared.

A typical abbreviated example of a QEP diagram involving a cart-prod is shown below:

```
          Cart-Prod
        /        \
    proj-rest    table
      /
  table
```

This node is displayed with the following information on a QEP diagram:

- A label identifying it as a cart-prod join node, along with the column(s) on which processing is done
- If an outer join has been requested, one of the following labels indicating the type of join:

  [LEFT JOIN]
  [FULL JOIN]
  [RIGHT JOIN]
- Storage structure (which is usually heap)
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

The cart-prod is most frequently observed in disjoint queries (that is, when use of correlation variables and table names are mixed in the query). However, the following cases can also generate cart-prods, without adversely affecting performance:

- Queries involving ORs that can usually be decomposed into a sequence of smaller queries
- No join specification (a disjoint table or no WHERE clause, so that there is no relationship between the two tables)
- Small tables or amounts of data
- Non_equijoins, such as a query with the following WHERE clause:

  ```
  where r1.col1 > r2.col2
  ```

Cart-prods are sometimes associated with substantial estimates for disk I/O usage and affect performance adversely.

This example shows a QEP diagram with a cart-prod join node resulting from the following simple disjoint query:

```
select arel.col1 from arel, arel a
  where a.col1 = 99;

QUERY PLAN 7,1, no timeout, of main query
                    Cart-Prod
                    Heap
                    Pages 1 Tups 243
                    D9 C4
                   /                \
          Proj-rest               Proj-rest
          Sorted(NU)              Heap
          Pages 1 Tups 2        Pages 1 Tups 156
          D1 C0                   D8 C1
          /                        /
     arel                      arel
     Hashed(col1)              Hashed(NU)
     Pages 70 Tups 156        Pages 70 Tups 156
```

Full Sort Merge Node

The *full sort merge* (FSM) is a more optimal join: it typically joins the inner and outer subtrees with many fewer comparisons than the cart-prod requires. This is done by assuring that both subtrees are sorted in the order of the join columns. If one or the other is not already sorted (for example, by being read from a B-tree index constructed on the join columns), the query plan can include a sort to put the rows in the correct order. This allows the rows of the outer subtree to be joined to the matching rows of the inner subtree with one pass over each. The inner subtree is not scanned multiple times, as with the cart-prod join.

A typical abbreviated example of a QEP diagram involving an FSM is shown below:

```
          join
        /    \
     sort    sort
      /       /
proj-rest   proj-rest
   /            /
table        table
```

This node is displayed with the following information on a QEP diagram:

- A label identifying it as a FSM join node, along with the column(s) on which join processing is done
- If an outer join has been requested, one of the following labels indicating the type of join:

  [LEFT JOIN]
  [FULL JOIN]
  [RIGHT JOIN]
- Storage structure (which is usually heap) or a list of sort columns if the data is being sorted
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

The FSM is most common when a "bulk" join takes place with no row restrictions on either table involved in the join, as with a SELECT statement of the following form:

```
select * from r1, r2 where r1.joincol = r2.joincol;
```

This example shows a QEP diagram with an FSM join node resulting from such a bulk join:

```
select a.col2, b.col2 from arel a, brel b
  where a.col1 = b.col1;
QUERY PLAN 5,1, no timeout, of main query
            FSM Join(col1)
            Heap
            Pages 1 Tups 156
            D9 C40
       /               \
    Proj-rest          Proj-rest
    Sorted(eno)      Sorted(eno)
    Pages 1 Tups 156   Pages 1 Tups 156
    D8 C1              D1 C1
   /                     /
arel                    brel
Hashed (NU)             Isam (NU)
Pages 70 Tups 156       Pages 3 Tups 156
```

Partial Sort Merge Node

The *partial sort merge* (PSM) is a cross between a full sort merge and a cart-prod. The inner tree must be in sorted order. The outer tree can be sorted or partially sorted. The outer tree in PSM scenarios can always be derived from an ISAM table. Comparisons proceed as for the full sort merge until an outer value is found to be out of order. At that point the inner loop is restarted. Because ISAM tables are reasonably well ordered (depending on how many rows have been added because the last reorganization), the number of inner loop restarts is typically small.

A typical abbreviated example of a QEP diagram involving a PSM is shown below:

```
          Join
       /      \
    proj-rest  sort
    /           /
  table     proj-rest
                /
              table
```

This node is displayed with the following information on a QEP diagram:

- A label identifying it as a PSM join node, along with the columns on which processing is done
- If an outer join has been requested, one of the following labels indicating the type of join:

  [LEFT JOIN]
  [FULL JOIN]
  [RIGHT JOIN]
- Storage structure (which is usually heap)
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

The following example shows a QEP diagram with a PSM join:

```
select a.col2, b.col2 from arel a, brel b
  where a.col1 = b.col2;

QUERY PLAN 6,1, no timeout, of main query
                     PSM Join(col1)
                     Heap
                     Pages 1 Tups 156
                     D9 C26
                    /          \
           Proj-rest            Proj-rest
           Heap                 Sort on(col1)
           Pages 1 Tups 156     Pages 1 Tups 156
           D1 C1                D8 C1
          /                     /
       brel                  arel
       Isam(col2)            Hashed(NU)
       Pages 3 Tups 156      Pages 70 Tups 156
```

Hash Join Node

The hash is an optimized join that replaces the FSM join when one or both of the subtrees must be sorted. It functions by loading the rows of one subtree into a memory resident hash table, keyed on the values of the join columns. The rows of the other subtree are hashed on their join key values into the hash table, allowing very efficient matching of joined rows. By avoiding the sort(s) of the FSM join, the hash join can be much more efficient.

A typical abbreviated example of a QEP diagram involving a hash join is shown below:

```
           Join
        /      \
    proj-rest  proj-rest
    /            /
  table     table
```

This node is displayed with the following information on a QEP diagram:

- A label identifying it as a hash join node, along with the columns on which join processing is done
- If an outer join has been requested, one of the following labels indicating the type of join:

  [LEFT JOIN]
  [FULL JOIN]
  [RIGHT JOIN]
- Storage structure (which is usually heap) or a list of sort columns if the data is being sorted
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

Like the FSM join, the hash join is most common when a bulk join takes place with no row restrictions on either table involved in the join, as with a SELECT statement of the following form:

```
select from r1,r2 where r1.joincol= r2.joincol;
```

This example shows a QEP diagram with hash join node resulting from such a bulk join:

```
select a.col2, b.col2 from arel a, brel b
where a.col1 = b.col2;

QUERY PLAN 1,1, no timeout, of main query
                     HASH Join(col1)
                     Heap
                     Pages 1 Tups 156
                     D9 C40
                    /             \
           Proj-rest            Proj-rest
           Heap                 Heap
           Pages 1 Tups 156     Pages 1 Tups 156
           D8 C1                D1 C1
          /                     /
       arel(a)                brel (b)
       Hashed (NU)           Isam (NU)
       Pages 70 Tups 156     Pages 3 Tups 156
```

Key and Tid Lookup Join Node

In *key* and *tid lookup joins*, the outer and inner data set is not static. For each outer row, the join selects values and forms a key to search in the inner join. A key lookup join uses keyed access through the structure of the inner table or index, and a tid lookup join uses the tuple identifier (tid) value. For more information, see Tids (see page 202).

A typical abbreviated example of a QEP diagram involving this type of join is shown below:

```
        Join
      /      \
   sort       btree (or hash or isam)
      \
    proj-rest
        \
      table
```

This node is displayed with the following information on a QEP diagram:

- A label identifying it as a key (K) or tid (T) lookup join, along with the column(s) on which processing is done
- If an outer join has been requested, the following label indicating the type of join:

  [LEFT JOIN]
- Storage structure (which is usually heap)
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

This case is seen most frequently where the outer subtree proj-rest returns few rows, so it is faster to do a keyed lookup (on an ISAM, hash, or B-tree table) than any of the sort merge operations.

The following example shows a QEP diagram with a key lookup join:

```
select b.col1, b.col2, a.col2
  from arel a, brel b
  where a.col3 = 'x' and a.col1 = b.col1;

                  K Join(col1)
                  Heap
                  Pages 1 Tups 2
                  D3 C0
                   /           \
           Proj-rest            brel
           Heap                Isam(col1)
           Pages 1 Tups 2       Pages 3 Tups 156
           D1 C0
            /
       arel
       Hashed(NU)
       Pages 70 Tups 156
```

In the next example of a tid lookup join, access to the base table is through the secondary index, and proj-rest collects tids for sorting. The tids are used for a direct tid lookup in the base table. Therefore, the storage structure of the base table is NU:

```
select a.col1, a.col2 from arel a
  where a.col2 = 99
  order by a.col2;

                   Sort(col2)
                   Pages 1 Tups 1
                   D4 C1
                   /
             T Join(tidp)
             Heap
             Pages 1 Tups 1
             D4 C0
            /               \
         Proj-rest          arel
         Sort on(tidp)      Hashed(NU)
         Pages 1 Tups 1     Pages 70 Tups 156
         D2 C0
        /
   aindex
   Isam(col2)
   Pages 2 Tups 156
```

Subquery Join Node

The *subquery join* is specific to SQL because SQL allows subselects as part of a query. These nodes join rows from a query to matching rows of a contained subselect, thus allowing the subselect restrictions on the query to be evaluated.

A typical abbreviated example of a QEP diagram involving a subquery join is shown below:

```
                SE join
              /      \
       proj-rest      Tn
         /
    table
```

In this diagram, T*n* identifies the QEP constructed to evaluate the subselect.

This node is displayed with the following information on a QEP diagram:

- A label identifying it as a subquery (SE) join, along with the column(s) on which processing is done
- Storage structure (which is usually heap)
- Total number of pages and tuples
- Query cost information
- Optionally, sort information (whether the data produced by this node is already sorted or requires sorting)

The following example shows a QEP diagram with a subquery join:

```
select * from arel a
  where a.col2 = (
    select col2 from brel b
      where a.col1 = b.col1)
  and col1 = 5;

QUERY PLAN 3,1, no timeout, of T1
               Proj-rest
               Heap
               Pages 1 Tups 1
               D1 C0
               /
          brel
          Hashed(col1)
          Pages 3 Tups 156
```

```
QUERY PLAN 4,2, no timeout, of main query
               SE Join(col1)
               Heap
               Pages 1 Tups 1
               D2 C0
              /              \
          Proj-rest          T1
          Heap               Heap
          Pages 1 Tups 1     Pages 1 Tups 1
          D1 C0
          /
      arel
      Hashed(col1)
      Pages 70 Tups 156
```

In the QEP pane of the SQL Scratchpad window in VDBA, these two QEPs are shown in separate tabs.

Subquery joins are reasonably expensive because the subselect must be executed once for each row of the outer subtree. The query optimizer contains logic to *flatten* various types of subselects into normal joins with the containing query. This allows more optimal joins to be used to evaluate the subselect.

As an example of the subselect flattening enhancement features of the query optimizer, consider the following subselect:

```
select r.a from r where r.c =
 (select avg(s.a) from s
    where s.b = r.b and r.d > s.d);
```

Instead of two scans of table r, the query optimizer has eliminated a scan of table r by evaluating the aggregate at an interior QEP node. The QEP appears similar to the previous example:

```
QUERY PLAN 7,1, no timeout, of main query
                     Hash Join(b)
                     avg(s.a)
                     Heap
                     Pages 2 Tups 359
                     D133 C171
                    /          \
             Proj-rest         Proj-rest
             Sorted(b)         Heap
             Pages 1 Tups 359  Pages 40 Tups 359
             D65 C112          D32 C3
            /                  /
          r                   s
          Btree (b,c)         Hashed(NU)
          Pages 44 Tups 359   Pages 44 Tups 359
```

Multiple Query Execution Plans

The query optimizer can generate multiple QEPs if the query includes any of the following objects:

- SQL subqueries (in, exists, all, any, and so on.)
- SQL UNION clause
- SQL GROUP BY clause
- Views that need to be materialized

As an example of multiple QEPs, consider the processing of a view on a union. The following statement creates the view:

```
create view view1 as
  select distinct col1 from arel
union
  select distinct col2 from arel;
```

There are two selects, designated #1 and #2 in their respective QEPs below. Now consider the query optimizer action in evaluating the following query:

```
select * from view1;
```

This generates three QEPs, which are shown in order in the example QEP diagrams that follow:

1. The first select in the union
2. The second select in the union
3. Main query—the merged result of the two unioned selects

```
QUERY PLAN of union view T0
               Sort Unique
               Pages 1 Tups 156
               D1 C10
              /
          Proj-rest
          Heap
          Pages 1 Tups 156
          D1 C0
          /
      aindex
      Isam(col2)
      Pages 2 Tups 156
```

```
QUERY PLAN of union subquery
             Sort Unique
             Pages 1 Tups 156
             D4 C10
            /
        Proj-rest
        Heap
        Pages 1 Tups 156
        D4 C0
       /
   arel
   Hashed(NU)
   Pages 70 Tups 156

QUERY PLAN of main query
             Sort Unique
             Pages 1 Tups 156
             D19 C20
            /
         Proj-rest
         Heap
         Pages 1 Tups 156
         D12 C11
        /
      T0
      Heap
      Pages 1 Tups 156
```

In the QEP pane of the SQL Scratchpad window in VDBA, these QEPs are shown in separate tabs.

More Complex QEPs

The previous series of QEPs on the different classes of joins involved only two tables. More complex QEPs involving joins with three or more tables can be read as a sequence of two-table joins that have already been described, with the query optimizer deciding what is the optimal join sequence. The key to understanding these complex QEPs is recognizing the join sequences and the types of joins being implemented.

Parallel Query Execution

With its thread support, Ingres has long supported the concurrent execution of separate queries. For short OLTP style queries, this permits a many fold increase in the number of such queries that can be executed in a given unit of time.

Ingres has the additional capability to split up the execution of individual long running queries over multiple threads. This parallel execution of a single complex query reduces the time to execute it.

Parallel query plans are implemented in Ingres by the introduction of the exchange node type. For more information, see Sample Parallel QEPs (see page 286).

An exchange node marks the boundary between processing threads in a query plan. It can spawn one or many threads to execute the query plan fragment below the exchange node concurrent with the fragment above the exchange node. The exchange node itself passes or exchanges rows from threads below the node to the thread above the node.

Types of Parallelism

Ingres compiles exchange nodes into queries to implement any of three types of parallelism:

- Inter-node (pipelined) parallelism – an exchange node that spawns a single thread effectively pipelines rows from the plan fragment below the node to the plan fragment above the node. For example, an exchange node below a sort node allows the plan fragment below to generate rows at the same time as sorting is being done for previous rows. Plan fragments that produce and consume rows at the same rate can effectively overlap their processing, reducing the overall execution time for the query.
- Inter-node (bushy) parallelism – exchange nodes inserted over essentially independent query plan fragments allow those fragments to execute independently of one another. A specialized case of bushy parallelism occurs in union queries when a single exchange node is placed above the unioned selects. One thread is created for each of the select plan fragments, allowing the selects to be processed concurrently.
- Intra-node (partitioned table) parallelism – a single exchange node is placed above the orig node for a partitioned table. The exchange node creates several (4, 8, and so forth) child threads, each one of which retrieves data from a subset of the partitions of the table. This allows the concurrent reading of rows from the different partitions, clearly reducing the elapsed time required to process the table. A variation on partitioned parallelism (called a partition compatible join) occurs when two partitioned tables with the same number of partitions are joined using their respective partitioning columns. The query optimizer places the exchange node above the join in the query plan, resulting in mini-joins being performed between the rows of compatible pairs of partitions. These mini-joins are performed concurrently on different threads.

Enabling Parallel Query Plans

The generation of parallel query plans is controlled by several configuration parameters, as well as a session level SET statement. The opf_pq_dop parameter defines the degree of parallelism or maximum number of exchange nodes that can be compiled into a query plan. A value of 0 or 1 prevents the generation of parallel plans, but any other positive value enables them. The session level set parallel <n> statement can be used to override the CBF parameter, where <n> is the degree of parallelism for queries compiled in the session.

The opf_pq_threshold parameter is a companion to opf_pq_dop and defines the cost threshold of a query plan before exchange nodes are inserted into it. Because there are slight overheads in initiating parallel query plans, compile only plans that benefit from parallel processing. The Ingres query optimizer currently compiles a serial query plan as it always has. If the degree of parallelism is defined to permit exchange nodes to be added to the plan, the cost estimate of the serial plan must still exceed the threshold value before a parallel plan is generated. The cost estimate is computed as the sum of the C and D numbers at the top of the query plan.

Sample Parallel QEPs

The following example shows a QEP diagram with a 1:n exchange node above a join node. It shows eight threads being created to divide the work of the join.

Two joins each of pairwise compatible partitions of the underlying tables are performed by each thread, all in parallel.

```
select a.col1, b.col2

   from aprel a, bprel b

   where a.col1 = b.col1

       and b.col2 between 500 and 1500
```

```
                                    Exchange
                                    Heap
                                    Pages 319 Tups 14012
                                    Reduction 4301
                                    Threads 8
                                    PC Join count 16
                                    D689 C6386
                         /
                        Hash Join(col1)
                        Heap
                        Pages 319 Tups 14012
                        PC Join count 16
                        D689 C6386
              /                     \
            Proj-rest               Proj-rest
            Heap                    Heap
            Pages 73 Tups 10000     Pages 83 Tups 14012
            D384 C100               D305 C140
 /                       /aprel           bprelB-Tree(NU)    B-Tree(col2)
Pages 768 Tups 10000    Pages 12872 Tups 299814
Partitions 128          Partitions 16
PC Join count 16        PC Join count 16
```

This example shows a union select in which the exchange node generates one thread to process each of the selects.

```
select a.col1 from arel a
    union all select b.col2 from brel b



QUERY PLAN 1,3, no timeout, of union subquery

            Proj-rest
            Heap
            Pages 25 Tups 8197
            D283 C82
 /
brel
(b)
Heap
Pages 1132 Tups 8197

QUERY PLAN 1,2, no timeout, of union subquery

            Proj-rest
            Heap
            Pages 16 Tups 5126
            D218 C51
 /
arel
(a)
Heap
Pages 872 Tups 5126

QUERY PLAN 1,1, no timeout, of main query

                        Exchange
                        Heap
                        Pages 54 Tups 17886
                        Reduction 242
                        Threads 2
                        D501 C266
             /
            Proj-rest
            Heap
            Pages 54 Tups 17886            D501 C266
 /
T3
Heap
Pages 16 Tups 17886
```

Optimizer Timeout

If the query optimizer believes that the best plan it has found so far takes less time to execute than the time it has taken evaluating QEPs, then it *times out*. It stops searching for further QEPs and returns the best plan is has found up to that point.

To tell if the query optimizer timed out, you can check either of the following:

- The QEP pane of the SQL Scratchpad window in VDBA. In the QEP pane, the Timeout check box is enabled if the optimizer timed out (and disabled if the optimizer did not time out).
- The header of the QEP for set qep diagrams. In QEP diagrams generated using set qep, the keywords "timed out" and "no timeout" are indicated in the QEP diagram header.

Note: The fact that the query optimizer has timed out on a particular query does **not** guarantee that a better QEP is found; it indicates that not all QEPs have been checked, and so potentially, a better QEP can be found.

Because it is elapsed CPU time that is being measured, QEPs that time out can change, depending on machine load.

Control Optimizer Timeout

By default, the optimizer times out, but you can turn the timeout feature off to enable the query optimizer to evaluate all plans.

To control optimizer timeout

To turn the timeout feature off, issue the following SQL statement (for example, from the VDBA SQL Scratchpad window, from a terminal monitor, or from within an embedded SQL application):

```
set joinop notimeout
```

To turn the timeout feature back on, issue the corresponding statement:

```
set joinop timeout
```

To control this feature using an operating system environment variable:

Windows:

```
set ING_SET=set joinop notimeout
```

Unix:

C shell:

```
setenv ING_SET "set joinop notimeout"
```

Bourne shell:

```
ING_SET = "set joinop notimeout"
```

```
export ING_SET
```

VMS:

```
define ING_SET "set joinop notimeout"
```

Greedy Optimization

The Ingres query optimizer performs an exhaustive search of all possible plans for executing a query. It computes a cost estimate for each possible plan and chooses the cheapest plan according to the cost algorithms.

Such a process is very fast for relatively simple queries. For queries involving large numbers of tables, however, the number of possible plans can be very large and the time spent enumerating the plans and associating costs with them can be significant.

While the Optimizer Timeout (see page 288) feature is useful in shortening processing time, the optimizer can take an unacceptable length of time to identify an efficient query plan in the case of very large queries, especially for queries with large numbers of potentially useful secondary indexes and queries whose tables have large numbers of rows (leading to very expensive plans that do not timeout quickly).

The query optimizer includes an alternative mechanism for generating query plans, called greedy optimization, that can greatly reduce the length of compile time. Rather than enumerate all possible query plans (including all permutations of table join order, all combinations of tables and useful secondary indexes, and all “shapes” of query plans), the “greedy” enumeration heuristic starts with small plan fragments, using them as building blocks to construct the eventual query plan. The chosen fragments are always the lowest cost at that stage of plan construction, so even though large numbers of potential plans are not considered, those that are chosen are also based on cost estimation techniques.

Note: The Optimizer Timeout feature does not work with greedy optimization. Unlike exhaustive enumeration, which constructs and costs whole query plans as it proceeds, greedy enumeration performs most of its work before it has any valid plan. Because of this, the optimizer timeout feature is ineffective and does not work with greedy enumeration. However, the speed of optimization using greedy enumeration is so great, there is no need for a timeout.

Control Greedy Optimization

By default, greedy optimization is used if the query meets the following two criteria:

1. The number of base tables is at least 5.
2. The combination of base table and potentially useful secondary indexes is at least 10.

For example:

Query	Greedy Used By Default?
7 tables and no indexes	No
3 tables and 7 indexes	No
7 tables and 3 indexes	Yes

The greedy heuristic typically chooses very good query plans, especially when considering the vastly reduced compile time. However, for the rare cases in which greedy optimization produces a much slower plan, it can be turned off.

To control whether greedy optimization is used

To turn greedy optimization off for the session, issue the following SQL statement (for example, from the SQL Scratchpad window in VDBA, from a terminal monitor, or from within an embedded SQL application):

```
[exec sql] set joinop nogreedy
```

To turn greedy optimization back on, issue the corresponding statement:

```
[exec sql] set joinop greedy
```

To turn off greedy optimization for the installation, set the opf_new_enum configuration parameter (in CBF or VCBF) to OFF.

Summary for Evaluating QEPs

The main points to check when evaluating a QEP are as follows:

- Cart-prods can be caused by errors due to disjoint queries or queries that involve certain types of OR operations. Also, joins involving calculations, data type conversions, and non-equijoins can generate cart-prods. Alternative ways of posing the query is often advised under these circumstances.
- The NU on the storage structure part in the orig node description is not a good sign if you believe the storage structure must have been used to restrict the number of rows being read.
- Verify that the appropriate secondary indexes are being used. Running the optimization process to generate statistics on the indexed columns allows the query optimizer to better differentiate between the selectivity powers of the different indexes for a particular situation.
- If there is little data in a table (for example, less than five pages) the query optimizer can consider a scan of the table rather than use any primary or secondary indexes, because little is to be gained from using them.
- Check that row estimates are accurate on the QEP nodes. If not, run the optimization process to generate statistics on the columns in question.

Specialized Statistics Processing

Optimizer statistics can be reviewed and processed by several utilities. You can:

- View and delete statistics
- Unload statistics to a text file
- Load statistics from a text file
- Copy a table and its associated statistics to another database
- Create sampled statistics

Display Optimizer Statistics

In VDBA, you use the Display Statistics dialog to view and delete statistics that have already been collected. For more information, see the online help topic Viewing Database Statistics.

You can also accomplish this task using the statdump system command. For more information, see the *Command Reference Guide*.

The usual output is based on statistics generated by the optimization process, as described in Database Statistics (see page 239).

Display Optimizer Statistics for Individual Tables and Columns

By default, optimizer statistics are shown for all tables and columns in the current database, but you can view statistics for specific table columns.

In VDBA, you use the Specify Tables and Specify Columns check boxes in the Display Statistics dialog. For example, specify that statistics be displayed only for the empno column of the emp table.

Delete Optimizer Statistics

You can delete statistics by enabling the Delete Statistics from Syscat check box.

Using this check box in conjunction with the Specify Tables and Specify Columns check boxes, you can specify the tables and columns for which to delete statistics.

For example, enable the Delete Statistics from Syscat check box, specify the empno and sex columns from the emp table and the empno column from the task table.

Floating Point Precision in Optimizer Statistics Display

You can specify the precision with which floating point numbers are displayed in the statistics by enabling the Set Precision Level to check box and entering a value in the corresponding edit control to determine the number of decimal digits in the text format of the floating point numbers.

For example, assume a table, t_float, is defined with a column named c_float of type float, and that the following statements are used to insert values (all of which are approximately 1.0):

```
insert into t_float values (0.99999998);
insert into t_float values (0.99999999);
insert into t_float values (1.0);
insert into t_float values (1.00000001);
insert into t_float values (1.00000002);
```

You can create statistics for this table using the optimization procedure described in Database Statistics (see page 239).

With its default floating point precision, the standard output is show seven places after the decimal point. For greater precision, you can enable the Set Precision Level check box and enter a larger value.

For example, specifying a precision level of 14 generates output similar to the following, in which there is sufficient precision to maintain a visible difference in the values:

```
*** statistics for database demodb version: 00850
*** table t_float rows:5 pages:3 overflow pages:0
*** column c_float of type float (length:8, scale:0, nullable)
date:2000_02_24 15:15:30 GMT  unique values:5.000
repetition factor:1.000 unique flag:Y complete flag:0
domain:0 histogram cells:10 null count:0.00000000000000     value length:8
cell:   0    count:0.00000000000000 repf:0.00000000000000 value:0.99999997999999
cell:   1    count:0.20000000298023 repf:1.00000000000000 value:0.99999998000000
cell:   2    count:0.00000000000000 repf:0.00000000000000 value:0.99999998999999
cell:   3    count:0.20000000298023 repf:1.00000000000000 value:0.99999999000000
cell:   4    count:0.00000000000000 repf:0.00000000000000 value:0.99999999999999
cell:   5    count:0.20000000298023 repf:1.00000000000000 value:1.00000000000000
cell:   6    count:0.00000000000000 repf:0.00000000000000 value:1.00000000999999
cell:   7    count:0.20000000298023 repf:1.00000000000000 value:1.00000001000000
cell:   8    count:0.00000000000000 repf:0.00000000000000 value:1.00000001999999
cell:   9    count:0.20000000298023 repf:1.00000000000000 value:1.00000002000000
```

This can be useful when statistics are output to a text file or input from a text file. For more information, see Statistics in Text Files (see page 294). When reading statistics from a text file, the optimization process assumes that all cell values are in ascending order. You can use the Set Precision Level option to preserve sufficient precision for floating point numbers.

Statistics in Text Files

The optimization process can directly read a set of optimizer statistics from an external text file, rapidly updating the statistics for specific database tables. This can be useful when:

- Information is being moved from one database to another (for example, using copydb), and you want to copy the statistics for the tables as well.
- You know the distribution of the data in a table and want to read or input these values directly, instead of letting the optimization process generate them for you.

The actual table data is ignored. This gives you a modeling ability, because the table can actually be empty but there are statistics that indicate the presence of data and its distribution. The query optimizer uses those false statistics to determine a QEP. For more information, see Query Execution Plans (see page 258). This gives the DBA the ability to verify correctness of QEPs without having to load data into tables.

The text file read by the optimization process can be created in one of two ways:

- When displaying statistics, you can unload statistics that already exist in the database and use the generated file as input to the optimization process.
- You can create an input file from scratch or by editing a file created when displaying statistics.

Unload Optimizer Statistics to a Text File

To unload optimizer statistics to a text file, you use the Direct Output to Server File option in the Display Statistics dialog in VDBA. The generated file is in an appropriate format so that it can be used as input to the optimization process. This allows:

- Statistics to be easily moved from one database to another
- A default text file to be created if you are generating your own statistics

For example, to dump all statistics from the current database into the file stats.out, enable the Direct Output to Server File check box and enter **stats.out** in the corresponding edit control.

Unload Statistics for Selected Tables or Columns

To unload statistics for selected tables or columns, use the Read Statistics from Server File option in conjunction with the Specify Tables and Specify Columns check boxes in the Display Statistics dialog in VDBA.

For example, if you want the stats.out file to contain statistics for the entire arel table and the col1 column in the brel table, enable the Specify Tables check box, and choose only the arel and brel tables from the Specify Tables dialog. Enable the Specify Columns check box and choose only the col1 column for brel from the Specify Columns dialog.

Sample Text File Statistics

A sample output file generated using the Direct Output to Server File option of the Display Statistics dialog is shown below. This same text file can be used as input to the optimization process, as described in the next section, Loading Optimizer Statistics from a Text File:

```
*** statistics for database demodb version: 00850
*** table brel rows:151 pages:3 overflow pages:1
*** column col1 of type integer (length:4, scale:0, nullable)
date:2000_02_24 16:04:37 GMT  unique values:132.000
repetition factor:1.144 unique flag:N complete flag:0
domain:0 histogram cells:16 null count:0.0000000       value length:4
cell:   0    count:0.0000000    repf:0.0000000    value:      0
cell:   1    count:0.0728477    repf:1.3750000    value:     23
cell:   2    count:0.0728477    repf:1.8333334    value:     31
cell:   3    count:0.0728477    repf:1.3750000    value:     59
cell:   4    count:0.0728477    repf:1.1000000    value:    138
cell:   5    count:0.0728477    repf:1.0000000    value:    151
cell:   6    count:0.0728477    repf:1.0000000    value:    162
cell:   7    count:0.0728477    repf:1.0000000    value:    173
cell:   8    count:0.0662252    repf:1.2500000    value:    181
cell:   9    count:0.0662252    repf:1.1111112    value:    193
cell:  10    count:0.0662252    repf:1.2500000    value:    202
cell:  11    count:0.0662252    repf:1.0000000    value:    214
cell:  12    count:0.0662252    repf:1.0000000    value:    224
cell:  13    count:0.0662252    repf:1.0000000    value:    236
cell:  14    count:0.0662252    repf:1.2500000    value:    256
cell:  15    count:0.0264901    repf:1.0000000    value:    261
```

Load Optimizer Statistics from a Text File

To load optimizer statistics from a text file, you use the Read Statistics from Server File option in the Optimize Database dialog. For example, if the file arelbrel.dat contains statistics for the arel and brel tables, these are loaded into the database by enabling the Read Statistics from Server File check box and entering **arelbrel.dat** in the corresponding edit control.

Load Statistics for Selected Tables or Columns

If the input file contains statistics for multiple tables, you can load selected tables or columns by using the Read Statistics from Server File option, in conjunction with the Specify Tables and Specify Columns check boxes in the Optimize Database dialog.

For example, if the file arelbrel.dat contains statistics for the arel and brel tables, just the statistics for arel are loaded into the database by enabling the Specify Tables check box and choosing only the arel table from the Specify Tables dialog.

To load only statistics for column col3 of the arel table, enable the Specify Columns check box and choose only the col3 column from the Specify Columns dialog.

Update Row and Page Counts

The input file for the optimization process contains information about the number of rows, as well as primary and overflow page counts in a table. However, because these values are critical to correct operation, these input values are normally disregarded when creating statistics, leaving the catalog values untouched.

You can force the values in the input file to be used when loading the statistics by enabling the Read Row and Page check box in the Optimize Database dialog.

Important! This option must be used with extreme care, because it sets critical values.

This option can be useful for certain specialized processing, such as query modeling and performance problem debugging. Bear in mind that the row count value can be modified for the table and its indexes. However, the page count is modified for the table only—the index page count values remains unchanged.

Copy a Table and Associated Statistics

You can copy a table and its associated optimizer statistics from one database to another using copydb and statistics that have been unloaded to a text file. This is usually much faster than copying only the table and rerunning the optimization process to recreate the statistics.

Note: Doing this makes sense only if the statistics are up-to-date.

First, unload the table and its statistics to text files, as described in the steps below:

1. Enter the following command to generate copy.in and copy.out scripts for the arel table:

   ```
   copydb olddb arel
   ```

2. Copy the are1 table out of the olddb database:

   ```
   sql olddb <copy.out
   ```

3. Use the Display Statistics dialog in VDBA to unload the statistics for the are1 table to a text file named are1.dat. For more information, see online help.

Next, copy the table and statistics back into the new database:

1. Copy the are1 table into the new database:

   ```
   sql newdb <copy.in
   ```

2. Use the Optimize Database dialog in VDBA to load the statistics for the are1 table from the text file are1.dat in Step 3 of the previous example. For more information, see online help.

Sampled Optimizer Statistics

The optimization process allows you to create sampled optimizer statistics on database tables. For large tables, sampled statistics are usually much faster to generate than full statistics, and if the percentage of rows to be sampled is chosen appropriately, they can be nearly as accurate.

Sampled statistics are generated by only looking at a certain percentage of the rows in a table. The percentage must be chosen so that all significant variations in the data distribution are likely to be sampled.

The sampled rows are selected by randomly generating values for the tuple identifier (tid), so tid values are required to support this functionality.

Create Sampled Statistics

To specify sampled statistics and the percentage of rows to be sampled, you use the Statistics on Sampled Data check box and the Percentage control in the Optimize Database dialog. For example, to optimize the table bigtable, sampling 3% of the rows, perform the following steps. For more information, see online.

1. Enable the Statistics on Sample Data check box.
2. Enter **3** for the Percentage.
3. Enable the Specify Tables check box.
4. Click Tables to open the Specify Tables dialog.
5. Enable the bigtable table, and click OK.
6. Click OK.

When sampling, the query optimizer chooses rows randomly from the base table and inserts them into a temporary table. Rows are selected in such a way that uniqueness of column values are preserved (conceptually, when sampling, a row can be selected not more than once). Full statistics, or minmax if requested, are created on the temporary table and stored in the catalogs as statistics for the base table. The temporary table is deleted. Be sure you have enough free disk space to store this temporary table, and that create_table permission has been granted to the user running the optimization process. For more information on granting privileges, see the *Security Guide*.

You have control over the percentage of rows that are sampled. It is worthwhile to experiment with this percentage. When the percentages are too small for a good sampling, the statistics created change as percentage figures change. As you increase the percentage, eventually a plateau is reached where the statistics begin coming out almost the same. The smallest percentage that provides stable statistics is the most efficient number.

Composite Histograms

Optimization is usually performed on individual columns. However, it is possible for Ingres to create and use histograms created from the concatenation of the key column values of a base table key structure or a secondary index. Such histograms are called composite histograms.

Composite histograms are useful in ad hoc query applications in which there are WHERE clause restrictions on varying combinations of columns. Such applications can have a variety of secondary indexes constructed on different permutations of the same columns with the goal of allowing the query optimizer to pick an index tailored to the specific combination of restrictions used in any one query.

For example, consider a table X with columns A, B, C, D, E, etc. and secondary indexes defined on (A, B, C), (B, C, D), (B, A, E). Consider a query with a WHERE clause such as "A = 25 and B = 30 and E = 99". With histograms on the individual columns, the Ingres query optimizer finds it difficult to differentiate the cost of solving the query using the (A, B, C) index and the (B, A, E) index. This is because of the technique used to determine the combined effect of several restrictions on the same table. However, with composite histograms defined on each index, the optimizer combines the three restrictions into a single restriction on the concatenated key values, and the (B, A, E) index clearly produces the best looking query plan.

Composite histograms can be created on the concatenated key values of a secondary index and on the concatenated key values of a base table index structure.

Chapter 12: Understanding the Locking System

This section contains the following topics:

Concurrency and Consistency

Ingres is a *concurrent* database system, which means it allows multiple users to access the same data at the same time.

In any database management system with multiple users, there is a trade-off between *concurrency* and *consistency*. Ideally, you want all users to be able to access any data at virtually any time (concurrency) but you must ensure that changes to the database are done in an orderly sequence that maintains the underlying structure of the data (consistency).

The task of the locking system is to manage access to resources shared by user databases, tables, and pages to guarantee the consistency of the shared data. Various types of *locks* are used to ensure that the database does not become inconsistent through concurrent accesses.

Locking System Configuration

The locking system works with the Ingres DBMS Server to coordinate access to databases.

The system administrator can initially configure the locking system during installation by setting parameters (typically system_lock_level and system_maxlocks). The locking parameters are installation-wide. They can be changed after installation only by the system administrator.

On UNIX systems, shared memory and semaphores are used as resources during lock control. The shared memory and semaphores used by your installation are configured in the operating system when the UNIX kernel is configured.

Lock Types

The locking system grants two types of locks:

Logical locks

Are held for the life of a transaction. The logical lock is held until you commit, roll back, or abort the transaction.

A *transaction* is a group of statements processed as a single database action and can consist of one or more statements.

Physical locks

Can be used and released in a transaction. The locking system uses them to synchronize access to resources.

Lock Modes

A lock has a *mode* that determines its power—for example, whether it prevents other users from reading, or only from changing, the data.

The six lock modes are as follows:

X

Exclusive locks or *write* locks. Only one transaction can hold an exclusive lock on a resource at any given time. A user of this lock is called a writer.

U

Update locks. Only one transaction can hold an update lock on a resource at any given time. This lock mode is used for update cursors. Update lock protocols are used by Ingres to increase concurrency and reduce deadlocks, because update locks can be converted to shared locks for rows and pages that are not updated.

S

Shared locks or *read* locks. Multiple transactions can hold shared locks on the same resource at the same time. No transaction can update a resource with a shared lock. A user of this lock is called a *reader*.

IX, IS

Intended exclusive and *intended shared* locks. Whenever the locking system grants an exclusive (X) or shared (S) lock on a page in a table, it grants an intended exclusive (IX) or intended shared (IS) lock on the table. An intended lock on a table indicates finer locking granularity (that pages in the table are being accessed). An IX lock indicates that pages are being updated, while IS indicates that pages are being read.

SIX

Shared intended exclusive locks. These locks specify a resource as "shared with intent to update." They can be considered as combining the functionality of S (shared) locks and IX (intended exclusive) locks. SIX locks are used in table locking strategies, where possible, to minimize the extent of exclusive locking required. The Ingres DBMS Server's buffer manager uses these locks to modify pages in its cache.

N

Null locks. These are locks that do not block any action but preserve the number in the value block of the locks or preserve data structures for further use.

Lock Levels

Locks can be of several *levels*. The level of a lock refers to the scope of the resource on which the lock is requested or used, for example, whether the lock affects:

- A single row
- A single page
- A table as a whole
- An entire database

The levels of locks subject to user control are row, page and table. Queries and commands use other lock levels that affect concurrency, such as database and table control locks.

Lock levels are as follows:

Row

Manages access to the row. Use row-level locking in situations where page locking can cause unnecessary contention or where increased concurrency is desired.

Row-level locking is not supported for tables that have a page size of 2 KB. Maxlocks is ignored with row-level locking (it only refers to page-level locks), but the number of row locks cannot exceed the maximum number of locks allowed per transaction, as specified by the system administrator when the locking system was configured. If it does, the row locks are escalated to a table-level lock. For more information, see Escalation of Locks (see page 309).

Page

Manages access to the data page, except by readers with the lockmode set to readlock = nolock. For more information, see Readlock = Nolock Option (see page 325).

Table

Manages all access to a table, except by readers with the lockmode set to readlock = nolock. For more information, see Readlock = Nolock Option (see page 325).

Database

Affects the ability of all users to connect to that database. A user blocked by an exclusive database lock is not able to connect to the database and receives an error message indicating that an exclusive lock is held.

Control

Manages a table while its schema is changed or loaded. This lock is always a physical lock.

For example, during the operation of data management utilities (create/drop table, create/drop index, create table as, modify and modify to relocate—or their equivalent operations in VDBA), an exclusive table control lock is used. This combination of mode and level of lock insures that no transaction can read a table while its schema is changed or loaded, even though the lockmode of the user is set to readlock = nolock. For more information, see Readlock = Nolock Option (see page 325). Conversely, readers can block data management utilities during schema read operations.

Value

When row locking, provides phantom protection for serializable users and serializes duplicate checking for unique indexes.

Note: We recommend that you be aware of what resources you lock during application development, database maintenance, and ad hoc queries.

For details on the set lockmode operation and user-controlled locking, see User-Controlled Locking (see page 318). For details on the SET LOCKMODE statement syntax, see the *SQL Reference Guide*.

How the Locking System Works

The locking system controls locking by doing the following:

- Managing and queuing lock requests
- Detecting deadlock situations

Lock Requests

When you either issue a statement or a command or perform the equivalent operation using VDBA, implicit requests for locks are made.

In addition, the locking system considers the following factors to determine what mode and level of lock, if any, to take:

- Are there any locks available in the system?
- Does the query involve reading or changing data?
- What resources are affected by the query?
- Are any other locks held on the affected resources?

Available Locks in the System

When the system administrator configures the logging and locking system, the total number of available locks is set. As each lock is used, a counter is decremented to reflect the number of locks still available. If a lock request is received after all available locks have been used, the request cannot be satisfied until a lock is freed.

If waiting for a free lock happens frequently, your system administrator can reconfigure the maximum number of locks (system_maxlocks parameter).

Lock Grants

Whether the locking system can grant a lock depends on whether another transaction holds a lock on the resource requested and, if so, what mode of lock that other transaction holds. If another transaction already holds an exclusive lock on the resource in question, a new lock cannot be used. The second request must wait.

The locking system uses intended shared and intended exclusive locks on a table to determine quickly whether a table-level lock can be used on that table, as follows:

- An intended shared lock on the table means that a shared lock has been used on at least one page of the table; nevertheless, a shared lock, if available, can still be used at either the page or table level.
- An intended exclusive lock on the table means that an exclusive lock has been used on at least one page of the table; no table-level lock can be used on the table on behalf of another user until the current page-level lock has been released.

Lock Mode Compatibility

The following table shows which granted lock modes are compatible with the requested mode:

	Granted Mode						
Req. Mode	NL	IS	IX	S	SIX	U	X
NL	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No	No
IX	Yes	Yes	Yes	No	No	No	No
S	Yes	Yes	No	Yes	No	No	No
SIX	Yes	Yes	No	No	No	No	No
U	Yes	No	No	Yes	No	No	No
X	Yes	No	No	No	No	No	No

For meaning of the lock mode abbreviations, see Lock Modes (see page 303).

How the Default Lock Mode is Determined

The locking system determines the default lock mode as follows:

- When you perform a read operation from the database, such as a select operation, a *shared lock* is requested on the affected resources for the transaction.
- When you perform an operation that writes to the database, such as an update, insert, or delete operation, the locking system requests an *exclusive lock* on the affected resources for the transaction. Update lock protocols are used by Ingres to increase concurrency and reduce deadlocks, because update mode locks can be converted to shared locks for rows and pages that are not updated. Update mode locks are converted to exclusive locks for rows and pages that are updated.

The default state of the locking system ensures that no user can read data being changed and no user can change data being read. However, users can read data that is being read by other users. This means that the locking system can grant an:

- S lock for User2 on resource R, provided User1 does not already hold an X lock on R.
- X lock on resource R for User2, provided User1 does not already hold an S or X lock on R.
- S lock on resource R for User2, even if User1 already holds an S lock on R.

This default strategy is adequate for most situations. When it is not, you can establish a different strategy using the SET LOCKMODE statement. For details, see User-Controlled Locking (see page 318).

How the Locking Level is Determined

By default, Ingres determines the best locking level to use. Ingres selects the locking level as page or table, depending on the optimizer's estimates of the number of pages to be read.

If the estimated number of pages to be read is greater than the session maxlocks limit, or if the entire table is to be read, Ingres uses table level locking. Otherwise, Ingres uses page level locking, and then escalates to table level locking if the number of page locks requested exceeds the session maxlocks per table, per query limit.

Initial Locking Level

In evaluating the query on which the lock is being requested, the Query Optimizer determines the level of lock as follows:

- If a query involves a single table with only a primary key, page-level locks are requested.
- If the optimizer estimates that no more than maxlocks pages are needed, the page-level locks are requested.
- If the Query Optimizer estimates that a query is touching more than the number of pages to which maxlocks is set, the query is executed with a table-level lock.
- If the Query Optimizer estimates that a query is touching all the pages in the table, the query is executed with a table-level lock.

This strategy saves the overhead of accumulating multiple page-level locks and prevents the contention caused by lock escalation. For example, on a query that is not restrictive or does not use a key to locate affected records, the locking system grants a table-level lock at the beginning of query execution.

Escalation of Locks

When page locking, if the number of pages in a table on which locks are held reaches maxlocks during a query, the locking system escalates to table-level locks. To do this it:

- Stops accumulating page-level locks
- Acquires an appropriate (S or X) table-level lock
- Releases all the page-level locks it has accumulated

The locking system also escalates to table-level locks in an attempt to complete a transaction if it exceeds the maximum number of locks allowed or the installation has run out of locks. If this occurs, an error is issued and the transaction is backed out. To avoid this situation in the future, the system administrator can bring down the installation and reconfigure the locking system.

Note: The issuing of lock escalation messages is configurable.

Methods for Changing How Locking is Handled

The following methods can be used to change how Ingres handles locking:

- Set the system_lock_level parameter. The system administrator can override the default locking level by setting the system_lock_level parameter. This parameter sets the default locking level for an entire Ingres instance.

  By default, system_lock_level is set to DEFAULT, in which Ingres decides the locking level. For details on the default behavior, see How the Locking Level is Determined (see page 308). Other valid values for system_lock_level are ROW, PAGE, and TABLE. Each of the default lock levels is subject to escalation. For example, if system_lock_level is set to PAGE, the default locking level is page, and then Ingres escalates to a table-level lock if the number of page locks requested exceeds the session maxlocks per table, per query limit.

- Use the SET LOCKMODE statement to change parameters that determine how locking is handled. For example, using maxlocks you can reset the maximum number of page-level locks that can be requested per table per query before escalating to a table-level lock.

  Note: The set lockmode statement cannot be issued in a transaction.

  For details, see User-Controlled Locking (see page 318).

Summary of Default Locks

The following table describes what mode and level of lock is invoked by default when a query is issued.

Statement or Command	Comment	Mode	Level
create index	On base table:	X	Table lock
		X	Control lock
	On index:	X	Table lock
create rule	On base table:	X	Table lock
create table	On table:	X	Table lock
		X	Control lock
create view	On view:	X	Table lock
		X	Control lock
	On base table:	X	Table lock
drop	On table:	X	Table lock
grant	On base table:	X	Table lock

Statement or Command	Comment	Mode	Level
modify	On table:	X X	Table lock Control lock
select	For each table involved in the select:	IS and S	Table lock Page lock(s) on pages in table
	If query touches 50 (or maxlocks) pages:	S S	Control lock Table lock
sysmod	On database:	X	Database lock
update, insert, or delete	On table involved in update, insert, or delete:	IX and X	Table lock Page lock(s) on pages in table
	If query touches 50 (or maxlocks) pages:	X	Table lock
	On other tables used in query but not being changed:	S	See lock for select statement

Releasing of Locks

A transaction accumulates locks on resources until you commit or roll back. When a transaction is committed, the results are written to the database, and all the locks accumulated during the transaction are released. A rollback aborts the transaction and releases accumulated locks.

You commit transactions during a session by doing one of the following:

- Issuing the COMMIT statement after one or more SQL queries
- Using the set autocommit on statement. This causes a commit to occur after every SQL query that was successfully executed during the session.

  For details on the set statement using the autocommit parameter, see the *SQL Reference Guide.*

After a commit is executed, the current transaction is terminated and you are in a new transaction as soon as the next SQL statement is issued.

All open transactions are automatically committed when you end your session.

Important! If you do not issue the commit statement during a session when the set autocommit is off, all locks requested on the resources affected by your queries are held until your session ends. Your entire session is treated like one transaction and can cause concurrency problems.

Example: Single User Locking

This example illustrates the use of locking when a single user initiates a transaction.

In this example, a user issues a single query transaction (SQT) consisting of:

- An SQL statement to read data on the employee named Jeff from the table named EMP
- A commit

Here is the sequence of operations:

1. The user issues a SELECT statement followed by a COMMIT statement:

   ```
   select * from emp where name = 'Jeff';
   commit;
   ```

2. An "IS" lock is used on the EMP table and a page-level "S" lock on the page containing the Jeff row.

   The query is restrictive (only the row specified in the WHERE clause is to be retrieved), and the table itself has an ISAM structure indexed on 'name', so the entire table does not have to be scanned. The locking system can use the index to go directly to the row for Jeff. Thus, an S lock on the entire table is not necessary; an IS table-level lock and an S lock on the page containing the row for Jeff are sufficient.

3. The Jeff row is retrieved.
4. The transaction is terminated and the locks held are released.

After retrieving the row for Jeff, if the user were to issue an UPDATE statement and change that record before issuing the commit, and if there were no other shared locks on the page, the locking system converts the shared page-level lock to an exclusive lock, and the IS lock on the table to an IX lock.

Example: Multiple User Locking

This example illustrates locking when multiple users initiate concurrent transactions against the same tables. In this case, users must wait for appropriate locks.

The first user (User1) initiates a transaction to update the salary of each employee in the Techsup department to 30000. Another user (User2) issues a query to read the salary and floor of the employee named Dan. Both users end their transactions with a COMMIT statement.

Both transactions affect the tables:

- Emp, which is keyed on *name* with a secondary index on *deptno*
- Dept, which is keyed on *dname*

Because of the way these tables are indexed, only a few pages in the tables need to be accessed, so page-level locking is used.

The following tables show the first four pages of the EMP and DEPT tables.

EMP Table

The EMP table is an ISAM structure keyed on the name column, and with a secondary index on the deptno column:

Page	Name	Salary	Deptno
1	Andy	55000	9
	Candy	50000	6
	Dan	25000	7
2	Ed	20000	2
	Fred	20000	8
	Jeff	35000	4
3	Kevin	40000	3
	Lenny	30000	6
	Marty	25000	8
4	Penny	50000	9
	Susan	20000	1
	Tami	15000	6

DEPT Table

The DEPT table is an ISAM structure keyed on the dname column:

Page	Deptno	Dname	Floor
1	1	Accting	5
	2	Admin	4
	3	Develop	4
2	4	Mgr	3
	5	Prod	2
	6	Sales	3
3	7	Shipping	2
	8	Techsup	1
4	9	VP	5
	10	WP	5

Locks Granted

The following shows the locks that are requested on behalf of both users:

Table	Page	User 1 Locks	User 2 Locks
Emp	Entire table	IX	IS
	1		S
	2	X	
	3	X	
	4		
Dept	Entire table	IS	IS
	1		
	2		
	3	S	S
	4		

Here is the sequence of operations:

1. User1 issues the following statements:

```
update emp set salary = 30000 where deptno in
    (select deptno from dept
    where dname = 'Techsup');
commit;
```

2. User2 issues the following statements:

```
select e.salary, d.floor from emp e, dept d
    where d.deptno = e.dept
    and e.name = 'Dan';
commit;
```

3. On behalf of User1, the following locks are requested:

   - An IS lock on the DEPT table
   - An S lock on the third page of the DEPT table where the record for the Techsup department is located

   The subselect statement starts executing, which retrieves the Techsup record.

4. On behalf of User2, the following locks are requested:

   - An IS lock on the EMP table
   - An S lock on the first page of the EMP table where the record for the employee named Dan is located
   - An IS lock on the DEPT table
   - An S lock on the third page of the DEPT table where the Shipping record is located

   The select statement starts executing, which retrieves the salary for employee Dan from the EMP table and the floor on which he works from the DEPT table, using the deptno value 7.

5. On behalf of User1, the following locks are requested:

   - An IX lock on the EMP table
   - An X on the second and third page of the EMP table where the updates is made

   The update statement starts executing, setting the value of the salary column for all employees in the Techsup department to 30000.

6. On behalf of User2, the commit statement is executed; releasing all locks held in User2’s behalf.

7. On behalf of User1, the commit statement is executed; committing all updates and releasing all locks held in User1’s behalf.

Waiting for Locks

If an IX lock is taken on a table on behalf of User1, User2 must wait to retrieve all the values from the table until User1 completes his transaction and releases all locks. This occurs because one user is updating at least one page in a table and the default is that no other user can read the entire table.

For example, assume that User2 in the previous example had issued the following statements instead:

```
select * from emp;
commit;
```

In this simple case, the waiting time is negligible, but had User1 issued a complicated update on a large number of rows in the EMP table, User2 must wait a long time.

Ways to Avoid Lock Delays

To prevent delays due to lock waits, there are several approaches:

- Keep transactions as short as possible. Use SET AUTOCOMMIT ON if possible.
- Set the READLOCK-NOLOCK lockmode when possible, to avoid waiting for shared locks.
- Use the SET LOCKMODE parameter timeout to indicate how long to wait for a lock. The default is to wait forever. If the timeout is reached, an error is returned and the current statement (not the transaction) is aborted. You must check for this error in your application code.

For details on the SET LOCKMODE statement, see User-Controlled Locking (see page 318).

User-Controlled Locking—SET LOCKMODE

User-controlled locking is available through the SET LOCKMODE option of the SET statement. This option provides the following types of locking parameters:

- Locking behavior
- Locking mode requested when reading data
- Maximum number of page locks
- Maximum length of time a lock request can remain pending

Important! You cannot issue the set lockmode statement in a transaction. You can issue it as the first statement in a session or after a COMMIT statement.

For details on the syntax for the SET LOCKMODE statement, see the *SQL Reference Guide*.

Ways to Specify a Set Lockmode Statement

There are several ways to specify the SET LOCKMODE statement:

- Issue the SET LOCKMODE statement from a terminal monitor, for example:

  ```
  set lockmode session where readlock = nolock;
  ```

- Include the SET LOCKMODE statement in an embedded language program. This affects only the session of the user issuing the statement.
- Specify SET LOCKMODE with any of the following environment variables or logicals:
  - ING_SYSTEM_SET—affects all users.
  - ING_SET—if set at the installation-wide level, affects all users. If set at the local level, affects the user who set it.
  - ING_SET_*dbname*—same as ING_SET but affects only the specified database.
  - *dbname*_SQL_INIT—affects only the SQL terminal monitor for the specified database. It can be set at the installation-wide level or the local level.

For example, to specify READLOCK = NOLOCK for your sessions with the SET LOCKMODE option using ING_SET, issue the following commands at the operating system prompt:

Windows:

```
set ING_SET="set lockmode session where readlock = nolock"
```

UNIX:

C shell:

```
setenv ING_SET "set lockmode session where readlock = nolock"
```

Bourne shell:

```
ING_SET = "set lockmode session where readlock = nolock"

export ING_SET
```

VMS:

```
define ing_set

  "set lockmode session where readlock = nolock"
```

The SET statements pointed to by the environment variables or logicals are executed whenever a user connects to the server. The environment variables or logicals can be set installation-wide or locally in each user's environment. For further details on setting these environment variables or logicals, see the *System Administrator Guide*.

Range of the Set Lockmode Statement

With the SET LOCKMODE statement, you can:

- Set locking parameters for a particular table. For example:

  ```
  set lockmode on emp where readlock = nolock;
  ```

- Set locking parameters for the duration of a session. For example:

  ```
  set lockmode session where readlock = nolock;
  ```

If multiple SET LOCKMODE statements are issued for the same session or table, the most recent statement is the one in effect. Setting a locking parameter on a specific table has precedence over the session setting. Any lockmode settings issued during a session end when that session ends.

When to Change the Locking Level

There are several situations where the page-level locking default is not appropriate:

- If a query is not restrictive or does not make use of the key for a table, scanning the entire table is required. In that case, the locking system automatically starts with a table-level lock; you do not need to specify it.
- If there are a number of unavoidable overflow pages, it is preferable to set table-level locking for reasons of efficiency.
- If, during execution of a query, more than maxlocks pages on a table must be locked (often because of an overflow chain), the locking system escalates to a table-level lock. It releases the page locks that have been accumulated. Because accumulating page locks when a table lock was really necessary is a waste of resources, table locking from the outset is preferable.
- If multiple users are concurrently running queries to change data, *deadlock* can occur.

  Deadlock occurs when multiple transactions are waiting for each other to release locks, and none of them can complete. For a discussion on deadlock, see Deadlock (see page 331).

  If page locking causes unnecessary contention, row-level locking can be used.

Change the Locking Level with Set Lockmode

To specify table-level locking, use the following statement:

```
set lockmode session where level = table;
```

To specify row-level locking, use the following statement:

```
set lockmode session where level = row;
```

The Maxlocks Value

By default, the locking system escalates to a table-level lock after locking 50 pages in a table during a query.

Note: Lock escalation can lead to deadlock.

By changing the value for maxlocks to a number greater than 50, you can reset the number of locks that are requested before escalation occurs.

Increasing this value requires more locking system resources, so the installation configuration for the maximum number of locks must be raised; but this can provide better concurrency in a table with unavoidable overflow chains.

Change Maxlocks Value with Set Lockmode

The following statement changes the number of pages in the EMP table that can be locked during a transaction from 50 to 70:

```
set lockmode on emp where maxlocks = 70;
```

With the new maxlocks value, the locking system escalates to a table-level lock only after more than twenty pages have been locked in table EMP during a query.

Timeout Value for a Lock Wait

By default, the locking system waits for a lock indefinitely. (The default is “0,” that is, no timeout.) For example, if User1 is running a report and User2 issues an INSERT statement for the table used for the report, the insert appears to “hang” while waiting for a lock. User2’s transaction waits for a lock on the table until User1’s report has completed, no matter how long that takes.

If you are not certain how long users in your database wait for locks, you need to limit the period of time (expressed in seconds) a user waits for a lock. This can be done using the timeout option of the SET LOCKMODE statement.

If a lock is not used in the amount of time specified, the statement is rolled back (not the entire transaction) and an error is returned. This error must be trapped and handled in embedded SQL and 4GL programs.

To immediately return control to the application when a lock request is made that cannot be granted without incurring a wait, use TIMEOUT=NOWAIT on the SET LOCKMODE statement.

Set a Timeout Value for a Lock Wait

To limit to thirty seconds the time that a lock request remains pending, issue the following statement:

```
set lockmode session where timeout = 30;
```

To immediately return control to the application when a lock request is made that cannot be granted without incurring a wait, issue the following statement:

```
set lockmode session where timeout = nowait
```

Guidelines for Timeout Handling

If you embed a SET LOCKMODE WITH TIMEOUT in an application, timeout must be carefully handled by the application. There are two cases, depending on whether cursors are used in the embedded application:

- No cursors—if a timeout occurs while processing a statement in a multiple query transaction, only the statement that timed out is rolled back. The entire transaction is not rolled back unless the user specifies rollback in the SET SESSION WITH ON_ERROR=ROLLBACK statement. For this reason, the application must be able to trap the error, and either re-issue the failed statement, or roll back the entire transaction and retry it starting with the first query. For more information on the SET SESSION statement, see the *SQL Reference Guide*.
- Cursors open—if one or more cursors are open when timeout occurs during a multiple query transaction, the entire transaction is rolled back and all cursors are closed.

We recommend that the timeout error handler check on the transaction status so it can tell which case was used. This can be done with an INQUIRE_SQL statement that retrieves the transaction state. For example, in the following statement xstat has a value of 1 if the transaction is still open:

```
exec sql inquire_sql (:xstat = transaction);
```

For a detailed description of the INQUIRE_SQL statement, see the *SQL Reference Guide*.

Example: Timeout Program

The following program example, written in ESQL/C and using the Forms Runtime System, checks for timeout and retries the transaction.

The program assumes an interface using a form to enter the department name, manager name, and a list of employees. The program inserts a new row into the department table to reflect the new department and updates the employee table with the new department name. An ESQL error handler checks for timeout. If timeout is detected, the user is asked whether to try the operation again.

```
/* Global variable used by main and by error handler */
int timeout;
main()
{
        int myerror();
        exec sql begin declare section;
                char deptname[25];
                char mgrname[25];
                char empname[25];
                char response[2];
        exec sql end declare section;
         . . .
        exec sql set lockmode session where timeout = 15;
        exec sql set_ingres(errorhandler=myerror);
         . . .
/* Assume this activate block starts a new transaction */
        exec frs activate menuitem 'addemp';
        exec frs begin;
                while (1)
                {
                        timeout=0;
                        exec frs getform empform (:deptname=dept, :mgrname=mgr);
exec sql insert into dept (dname, mgr)
        values (:deptname, :mgrname);
                        if (!timeout)
                        {
                                exec frs unloadtable empform emptbl (:empname=name);
                                exec frs begin;
                                        exec sql update emp set dept = :deptname
                                                where ename = :empname;
                                        if (timeout)
                                                exec frs endloop;
                                                /* Terminate unloadtable */
                                exec frs end;
                        }
                        if (!timeout)
                        {
                                exec sql commit;
                                break;
                        }
                        else
                        {
                                exec sql rollback;
                                exec frs prompt ('Timeout occurred. Try again? (Y/N)',
                                        :response);
                                if (*response == 'N')
                                        break;
                        }
                }
        exec frs end;
         . . .
}
int
myerror()
{
#define TIMEOUT 4702
        exec sql begin declare section;
                int locerr;
        exec sql end declare section;
        exec sql inquire_sql (:locerr = dbmserror);
        if (locerr == TIMEOUT)
                timeout = 1;
}
```

Readlock Option

Pages locked for reading are normally locked with a shared lock. A shared lock on a page does not prevent multiple users from reading that data concurrently.

However, a user trying to change data on the locked page must wait for all shared locks to be released, because changing data requires exclusive locks.

This can be a problem if one user is running a long report that accesses a table with a shared lock. No users can make changes to the locked table data until the report is complete.

Readlock=Nolock Option

Setting the lockmode to READLOCK=NOLOCK on a table accessed by a user running a long report allows others users to modify the table data while the report is running. Using READLOCK=NOLOCK does not affect any query that updates, deletes, or inserts rows in a table.

Note: If READLOCK=NOLOCK is set, and rows are changed while a report is being run on the table, the report is not a consistent snapshot of the table. Before using this strategy, consider the importance of the consistency and accuracy of the data.

A table control lock is used to ensure that no reader of any type (including when READLOCK=NOLOCK is set) can look at a table when:

- It is being loaded using the COPY or the CREATE TABLE...AS SELECT statement
- Its schema is being created or changed, using any of the following statements; a READLOCK=NOLOCK reader blocks the following operations:
  - CREATE TABLE
  - CREATE INDEX
  - CREATE VIEW
  - CREATE INTEGRITY
  - DROP
  - MODIFY

Whereas shared locks prevent other users from obtaining write locks and slow down their performance, setting READLOCK=NOLOCK can improve concurrent performance and reduce the possibility of deadlocks.

Set Readlock to Nolock

To set READLOCK to NOLOCK, issue the following statement:

```
set lockmode session where readlock = nolock;
```

When Readlock=Nolock is Beneficial

Setting readlock to nolock is beneficial when:

- Running a report to get an overview of the data, and absolute consistency is not essential.
- Updates, inserts, or deletes to a table involve isolated operations on single rows rather than multiple query transactions or iterative operations on multiple rows.
- Reports are needed on tables that are being concurrently updated. Reports slow down the updates and vice versa. Setting readlock = nolock on the reporting sessions improves concurrency. (If the report must provide a consistent snapshot, it is preferable to set readlock = exclusive and get the report done quickly.)
- Running reports "in batch" with a low priority. Running reports this way causes the locking of tables and pages for extended periods because of the lower priority. Setting readlock = nolock allows reporting to run at a low priority without disrupting other online users.

When Readlock=Nolock is Undesirable

Setting readlock to nolock is undesirable when:

- Other users are doing updates that use multiple query transactions or iterative operations (for example, increase all salaries by 10%), yet it is necessary that a report accurately take a "snapshot" of the table, either before or after the complete transaction has taken place.
- Using multiple query transactions that include updates that reference data from other tables. Here you cannot guarantee the consistency of data between the tables with readlock = nolock.

Readlock=Exclusive Option

A locking option that is useful in special circumstances is setting readlock to an exclusive lock.

Here is an example where controlling the shared lock locking level is necessary. User1 submits a multiple query transaction that retrieves data into a table field that the user is allowed to change before writing changes back into the table. At the same time, User2 submits a multiple query transaction to retrieve the same set of data into his or her table field, makes changes to the data, and writes the changes back to the table.

Eventually, the two users deadlock. Each is waiting for the other to finish and release the shared lock, so that each one can get an exclusive lock to make changes.

If the retrievals and changes had not been done with a multiple query transaction, no deadlock has occurred, because the shared locks are released before the requests for exclusive locks are made. But the exclusive lock transaction is necessary to prevent data from changing between the times you read the data and write to it.

It is preferable to exclusively lock the data when reading it into the table field, so that no other user can also retrieve the same set of data until the first user is finished. This can be achieved by setting exclusive readlock.

If it is likely that User1 holds these locks for a long time after retrieving into the table field and before committing changes, set timeout. For this reason, changing data inside a multiple query transaction is discouraged.

Set Readlock=Exclusive

To set READLOCK to EXCLUSIVE, using the following statement:

```
set lockmode session where readlock = exclusive;
```

Isolation Levels

Isolation levels allow users to specify an appropriate compromise between consistency and concurrency. This feature makes it possible to increase concurrency when the absolute consistency and accuracy of the data is not essential.

Ingres supports four isolation levels defined by the ANSI/ISO SQL92 standard:

- Read Uncommitted (RU)
- Read Committed (R)
- Repeatable Read (RR)
- Serializable

The highest degree of isolation is called “serializable” because the concurrent execution of serializable transactions is equivalent to a serial execution of the transactions. Serializable execution is the default behavior of Ingres transactions because it offers the highest degree of protection to the application programmer. This highest degree of isolation, however, is the lowest degree of concurrency.

At lower degrees of isolation, more transactions can run concurrently, but some inconsistencies can occur.

An isolation level is set by using the SET SESSION ISOLATION LEVEL and SET TRANSACTION ISOLATION LEVEL statements.

Inconsistencies During Concurrent Transactions

The ANSI/ISO specifies three inconsistencies that can occur during the execution of concurrent transactions:

- Dirty Read—transaction T1 modifies a row. Transaction T2 reads that row before T1 performs a commit. If T1 performs a rollback, T2 reads a row that was never committed and is considered to have never existed.
- Non-repeatable Read—transaction T1 reads a row. Transaction T2 modifies or deletes that row and performs a commit. If T1 attempts to reread the row, it can receive the modified value or discover that the row has been deleted.
- Phantom Rows—transaction T1 reads the set of rows N that satisfy some search condition. Transaction T2 executes SQL statements that generate one or more rows that satisfy the search condition used by transaction T1. If transaction T1 repeats the initial read with the same search condition, it obtains a different collection of rows.

Inconsistencies and Isolation Levels

The following table shows how the ANSI standard defines which inconsistencies are possible (Yes) and impossible (No) for a given isolation level:

Isolation Level	**Dirty Read**	**Non-Repeatable Read**	**Phantom Rows**
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

For programmers who are aware of possible inconsistencies, lower degrees of isolation can dramatically improve throughput.

The most commonly cited example of this is a cursor-based program that scans through a large table, examining many rows, but updating only a few rows. Under normal serializable execution, this transaction takes share locks on all rows or pages that it reads—typically, it takes a shared lock on the entire table—thus locking out all update activity on the table until the transaction commits or aborts.

Read Uncommitted Isolation Level

The Read Uncommited (RU) isolation level provides greatly increased read and write concurrency, but it suffers from the “dirty read” anomaly. Greater concurrency is achieved because the RU transaction does not acquire locks on data being read and other transactions can immediately read or/and modify the same rows. RU is ideal for applications where the reading of uncommitted data is not a major concern.

Read Committed Isolation Level

The Read Committed (RC) isolation level is well suited to allowing increased concurrency that is more controlled than at the RU level. RC transactions do not perform dirty reads but rather hold a lock on data while reading the data. For "cursored" queries, a lock is held on the current data item (page or row) pointed to by the cursor. The lock is automatically released when the cursor is positioned to the next item or closed. However, if any data on the current item of the cursor is changed, the lock must be held until the transaction commits. Such locking strategy is called *cursor stability*, and it defines an isolation level slightly stronger than "classical" RC.

The reason for cursor stability at the RC isolation level is to prevent cursor lost updates that are possible if locks are released immediately after data is read. The problem occurs when a transaction T1 running at the "classical" RC isolation level reads a data item; transaction T2 updates the data item and commits; T1 updates the data based on its earlier read value and also commits. T2's update is lost! Because of cursor stability, this problem does not exist in Ingres at the RC and higher isolation levels. At the same time, the RC mode does not guarantee that a transaction sees the same data if it repeats the initial read.

Cursor stability assumes that whenever the user is accessing a row with a cursor, this row is locked. However, if the user issues a complex cursor declaration that involves a join, and the results of the join are placed into a temporary buffer to be sorted before being updated, the assumption can be wrong. The problem exists because, in this case, the FETCH statement returns rows to the user, not from the base table, but from the temporary buffer. When the user attempts to update the "current" row of the cursor, the server locates the proper row of the base table by its TID taken from the temporary buffer. The user expects a lock to be held on the base table row until the row has been processed, but at the RC isolation level, the lock is released when the row is placed into the temporary buffer. Therefore, the row to be updated no longer exists or no longer meets the criteria in the WHERE clause. To prevent this problem, the server automatically upgrades the isolation level from "RC" to "RR" when the query is initiated.

Repeatable Read Isolation Level

In Repeatable Read (RR) isolation mode, locks are automatically released from data opened for reading but never read. With this option, if the application process returns to the same page and reads the same row again, the data cannot have changed. At the same time, repeatable read does not prevent concurrent inserts: if the same SELECT statement is issued twice (in the same transaction), "phantom rows"can occur.

Serializable Isolation Level

The Serializable isolation mode requires that a selected set of data not change until transaction commit. The page locking protocols prevent phantoms because the page locks cover the pages that hold the phantom. Simple row-level locking can provide repeatable read, but preventing phantoms in the serializable mode requires extra locks. These locks include data page locks for the ISAM and heap tables, value locks for the hash table, and leaf page locks for the B-tree table.

An isolation level is automatically increased from RC and RR to serializable for any operation on system catalogs and during the checking of integrity constraints or the execution of actions associated with referential constraints. This is necessary to ensure data integrity. However, if an integrity constraint is implemented by a user-defined rule, it is the user's responsibility to provide the appropriate isolation level.

Deadlock

Deadlock is a different condition than waiting for a lock. It occurs when one transaction is waiting for a lock held by another transaction **at the same time** that the other transaction is waiting for a lock held by the first. Both transactions block each other from completing. One of the transactions must be aborted to break the deadlock and allow the other to proceed.

Deadlock should be avoided.

Deadlock Example

This example (where the SET AUTOCOMMIT option is off) depicts a situation that produces deadlock.

User1 initiates a multiple query transaction to read all the data from the employee table and insert a record with the department name Sales into the DEPT table. Shortly after, User2 initiates a multiple query transaction to read all the data from the DEPT table and to insert a record with the employee name Bill into the EMP table.

Here is the sequence of operations:

1. User1 issues the statement:

   ```
   select * from emp;
   ```

2. On behalf of User1's transaction, a shared lock is requested on the EMP table and execution of the SELECT statement begins.
3. User2 issues the statement:

   ```
   select * from dept;
   ```

4. On behalf of User2's transaction, a shared lock is requested on the DEPT table and execution of the SELECT statement begins.
5. User1 enters the following statement:

   ```
   insert into dept (dname) values 'Sales';
   ```

6. User2 enters the following statement:

   ```
   insert into emp (name) values 'Bill';
   ```

7. User1's implicit request for an IX lock on the DEPT table is blocked because there is a shared lock on the table.
8. User2's implicit request for an IX lock on the EMP table is blocked because there is a shared lock on the table.

User1's transaction must wait for User2's transaction to release the shared lock on the department table, but this can never happen unless User2's transaction can finish. To finish, User2's transaction needs to obtain an exclusive lock on the employee table, which it cannot get until User1's transaction releases its shared lock on it.

Thus, both transactions are waiting for each other. Neither transaction can finish until the locking system checks on all transactions waiting for locks to make sure deadlock has not occurred.

When a deadlock is discovered, the locking system aborts one of the transactions, allowing the other transaction to continue. The user whose transaction was aborted receives an error.

All updates made by the transaction are backed out. For this reason, the deadlock error must be trapped and the transaction retried in an application program.

Deadlock does not occur frequently if transactions are concise and no lock escalation occurs (either page to table or shared lock to exclusive lock). A deadlock is always logged to the error log.

Deadlock in Single Query Transactions

Because the locking system uses page-level locking, accumulating locks one by one, deadlock can occur even when single query transactions are being used. At least two transactions must be accessing the database, and at least one user must be modifying rows. Deadlock does not occur when only SELECT statements are executing, because shared locks do not conflict with each other.

It is possible for deadlock to occur during a single query transaction when:

- Different access paths to pages in the base table are used
- Lock escalation occurs

Lock escalation deadlock can be caused by any of the following:

- Converting shared lock to exclusive lock
- Overflow chains
- System lock limits exceeded
- maxlocks exceeded
- B-tree index splits

Different Access Paths as a Source of Deadlock

Multiple transactions updating table data using different access paths can cause single query deadlocks.

Consider the following example in which the EMP table has an ISAM structure indexed on name and a hash secondary index on empno.

1. User1, accessing the EMP table through the secondary index, grants an exclusive lock on the fourth page of the table.
2. User2, accessing the EMP table by way of the ISAM key on the base table, grants an exclusive lock on the third page.
3. User1 needs an exclusive lock on the third page, but cannot get one because User2 already has a lock on it.
4. User2 needs an exclusive lock on the fourth page, but cannot get one because User1 already has a lock on it.

Lock Escalation as a Source of Deadlock

When multiple transactions are updating a table, and lock escalation occurs, they can deadlock. This escalation is probably caused by one of three situations:

- A transaction has run into a lock limit and can only continue by escalating to table-level locks.
- More than maxlocks pages need to be locked during the course of a query.
- There are long overflow chains.

If you are running into locking limits, either raise these limits or shorten the multiple query transactions.

If lock escalation deadlock is occurring frequently, consider using the SET LOCKMODE statement to force table-level locking on the table or to increase maxlocks.

To understand how lock escalation can produce deadlock, consider the following example in which two users are trying to insert into the same table that has many overflow pages:

User1 tries to insert a record, and because of the long overflow chain exclusively locks ten pages. Meanwhile, User2 also tries to insert a record and grants locks down another overflow chain.

During the processing of User1's query, the transaction reaches maxlocks pages and needs to escalate to an exclusive table-level lock; but, because User2 still holds an intent exclusive (IX) lock on the table, User1's request must wait.

User2's query also needs to lock more than maxlocks pages, so a request is made to escalate to an exclusive table-level lock. User2's request is also blocked, because User1 is holding an intent exclusive (IX) lock on the table.

Deadlock occurs in that neither user can proceed because each is blocking the other.

When many concurrent users are inserting into a small B-tree table, index splits are likely to occur and deadlock can occur because the locking level in the index must be escalated to exclusive.

Overflow Chains and Locking

Tables with excessive overflow pages can cause locking problems because all overflow pages must be searched. Each page is locked individually and locks are kept all the way down the overflow chain. Escalation to table-level locking while locking an overflow chain can cause deadlock in heavily concurrent environments, as well as slow down the query processing time. If you have a table with many unavoidable overflow pages (that is, they are still present after a remodify), use the SET LOCKMODE statement to do the following:

- Establish table-level locking as the default for that table
- Increase maxlocks

Deadlock in Applications

The following program sample checks for deadlock after each statement of a multiple query transaction. If deadlock occurs when a statement is issued, and that statement is the victim, the entire transaction containing the statement aborts and the application is sent back to the beginning of the transaction, where it is retried until it completes without deadlock.

This sample program is written in embedded SQL/Fortran:

```
exec sql include SQLCA;
exec sql whenever sqlerror goto 100;
exec sql whenever not found continue;
exec sql begin declare section;
       integer*4 x;
exec sql end declare section;

x = 0;

exec sql commit;

10  continue;

exec sql select max(empno) into :x from emp;
exec sql insert into emp (empno) values (:x + 1);
exec sql commit;

goto 200;

100  if (sqlcode .eq. -4700) then goto 10
        endif

200
  .
  .
```

In this example, if deadlock occurs, there is no need to issue the rollback statement, because the transaction has already been aborted.

If deadlock was not checked for and handled, and the select statement to retrieve the maximum employee number failed with a deadlock, the program flow continues and the next statement issued, the insert statement, is completed:

```
insert into emp (empno) values (:x + 1)
```

Because the select statement did not complete, this statement inserts the value "1," which probably is not the maximum employee number.

The default behavior in embedded SQL programs is to continue when an error occurs, and that errors are not printed by default. To handle an error, you need to specify the desired behavior in the whenever sqlerr statement or to check the sqlca.sqlcode manually after each SQL statement.

Ingres 4GL provides the while and endloop statements that perform the function of a goto statement and allow for checking and handling of deadlock. The following is an example of Ingres 4GL:

```
initialize(flag=integer2 not null,
        err=integer2 not null) =
{
}
'Go' = {
        flag := 1;
        a: while 1=1 do
        b: while flag=1 do
                repeated update empmax
                        set maxempno=maxempno + 1;
                inquire_ingres (err = errno);
                if err = 49900 then
                        endloop b; /* jump to endwhile of loop b */
                endif;
                repeated insert into emp (empno)
                        select maxempno from empmax;
                inquire ingres (err = errorno);
                if err = 49900 then
                        endloop b; /* jump to endwhile of loop b */
                endif;
                flag := 0; /*resets flag if MST successful */
                endwhile; /* end of loop b */
        if flag = 0 then
                commmit
                endloop a; /* jump to endwhile of loop a */
        endif;
        endwhile; /* end of loop a */
}
```

Tools for Monitoring Locking

You can identify problems with concurrency using one of the following lock monitoring tools:

- The Performance Monitor utility in VDBA, which displays locking data in a GUI environment
- The lock_trace trace flag, which displays specific locking activity
- The lockstat utility, which provides a summary listing and a "snapshot" of all of the locking activity in your installation
- The Interactive Performance Monitor (IPM), which provides locking data in a forms-based monitoring utility

Performance Monitor

The Performance Monitor utility allows you to view locking information in an easy-to-use GUI environment. By clicking on the Locking System branch in the window, you can immediately see a summary of the locking system information in the Detail pane.

Locking information you can view in the Performance Monitor includes:

- Lock lists
- Locked resources (databases, tables, pages, and others)
- Information about a lock (including the lock list, server, session, and resource of the lock)

The navigational tree in the left pane allows you to drill down to the information you need quickly, making it easy to identify locking conditions that need attention.

VDBA provides an alternative set of system administration tools, including monitoring performance. For instructions on using VDBA screens to monitor performance, see VDBA online help.

For more information on using the Performance Monitor utility, see the *System Administrator Guide*.

Set lock_trace Statement

The SET LOCK_TRACE statement enables you to start and stop lock tracing at any time during a session. This statement has the following syntax:

```
set [no]lock_trace
```

Important! Use SET LOCK_TRACE as a debugging or tracing tool only. The LOCK_TRACE option is not a supported feature. This means that you must not include this feature in any application-dependent procedure.

To use SET LOCK_TRACE you can:

- Issue the SET LOCK_TRACE statement from a terminal monitor. For example, to start tracing locks, issue the following statement:

  ```
  set lock_trace;
  ```

  To stop tracing locks, issue the following statement:

  ```
  set nolock_trace;
  ```

- Include the SET LOCK_TRACE statement in an embedded language program.
- Specify SET LOCK_TRACE with an environment variable or logical. For example, to start lock tracing with ING_SET issue the following statement at the operating system prompt:

  Windows:

  ```
  set ING_SET=set lock_trace
  ```

  UNIX:

  C shell:

  ```
  setenv ING_SET "set lock_trace"
  ```

  Bourne shell:

  ```
  ING_SET="set lock_trace"
  export ING_SET
  ```

  VMS:

  ```
  define ing_set "set lock_trace"
  ```

The same methods are used for SET LOCKMODE. For details on these methods, see Ways to Specify a Set Lockmode Statement (see page 318).

When you use SET LOCK_TRACE during a session, you receive information about locks used and released by your statements. This information is displayed with the results of your statement.

If you use an environment variable/logical to set the LOCK_TRACE flag, you receive output for utility startup queries as well as for query language statements.

lock_trace Output

An example of lock_trace output is shown here. The column headings above the example are added in this guide to help describe the output.

```
Action   Level   Qual.  Mode      Timeout       Key

LOCK:    PAGE    PHYS   Mode: S   Timeout: 0    Key: (inv,iiattribute,21)
UNLOCK:  PAGE    Key: (inv,iiattribute,21)
LOCK:    PAGE    PHYS   Mode: S   Timeout: 0    Key: (inv,iiindex,11)
UNLOCK:  PAGE    Key: (inv,iiindex,11)
LOCK:    TABLE   PHYS   Mode: IS  Timeout: 0    Key: (inv,parts)
LOCK:    PAGE           Mode: S   Timeout: 0    Key: (inv,parts,0)
```

The lock_trace output is in the following format:

```
action  level  qualifiers  Mode:  Timeout:  Key:
```

where:

action

Is the action, which can be LOCK, UNLOCK, or CONVERT. For example, a lock was used (LOCK) or released (UNLOCK).

level

Is the lock level, which can be TABLE, PAGE, ROW, or VALUE.

Other strings may appear, such as SV_PAGE or BM_DATABASE, which are internal cache control locks.

qualifiers

Specify more information about the lock. The qualifier can be:

NOWT—Do not wait if the lock is unavailable.

PHYS—Lock can be released prior to end of transaction (physical lock).

Blank—Lock is held until the transaction commits or aborts (logical lock).

Other qualifiers that may appear have internal meaning only.

Mode

Is the lock mode. Values can be:

S = shared lock

U = update lock

X = exclusive lock

IS = intended share

IX = intended exclusive

N = null lock

SIX = shared intended exclusive

Timeout

Is the default timeout or the timeout set with SET LOCKMODE statement.

Key

Describes the resource being locked. It consists of the database name, table name, partition and page number (shown as P.p where P is the physical partition number, and p is the page number), and (for row locking) the row number.

For VALUE level locks, the Key is database name, table name, and three numbers describing the value being locked. If the table is partitioned, the table name may be shown as an internal partition name, which looks like "iiXXX ppPPP-table name" where XXX is an internally assigned number, and PPP is the physical partition number. For example:

```
LOCK:  TABLE PHYS Mode: IX  Timeout:   0 Key: (emp,ii119 pp2-range_1)
```

lock_trace Example

The set lock_trace output for the following transaction is shown here.

```
select * from parts where color = 'red';
update parts set price = 10 where partno = 11;
commit;
```

This guide numbers the lines of output in the example. Each line number is explained.

Note: If you run the same query several times, you begin to receive less set lock_trace output because the system catalog information is being cached.

```
select * from parts where color = 'red'

+------+-------------+------+-----------+-----+
|partno|partname     |color |wt         |price|
 ---------------------------------------------

*********************************************************************************
(1) LOCK:   PAGE   PHYS  Mode: S  Timeout: 0 Key: (inv,iirelation,11)
(2) LOCK:   PAGE   PHYS  Mode: S  Timeout: 0 Key: (inv,iiattribute,21)
(3) UNLOCK: PAGE   Key:  (inv,iiattribute,21)
(4) LOCK:   PAGE   PHYS  Mode: S  Timeout:   Key: (inv,iiattribute,19)
(5) UNLOCK: PAGE   Key:  (inv,iiattribute,19)
(6) UNLOCK: PAGE   Key:  (inv,iirelation,11)
(7) LOCK:   PAGE   PHYS  Mode: S  Timeout: 0 Key: (inv,iiindex,11)
(8) UNLOCK: PAGE   Key:  (inv,iiindex,11)
(9) LOCK:   TABLE  PHYS: Mode: IS Timeout: 0 Key: (inv,parts)
(10)LOCK:   PAGE         Mode: S  Timeout: 0 Key: (inv,parts,0)
*********************************************************************************
|1A12 |Truck        |red     |290.000   | $16.00 |
|1B5  |Bean bag     |red     |198.000   | $18.00 |
|20G  |Laser        |red     |165.000   | $15.80 |
+-----+-------------+--------+----------+--------+

(3 rows)

update parts set price = 10 where partno = 20G
*********************************************************************************
(11)LOCK:   TABLE   PHYS Mode: IX  Timeout: 0 Key: (inv,parts)
(12)LOCK:   PAGE         Mode: U   Timeout: 0 Key: (inv,parts,0)
(13)LOCK:   PAGE         Mode: X   Timeout: 0 Key: (inv,parts,0)
*********************************************************************************

(1 row)

commit
*********************************************************************************
(14)UNLOCK: ALL     Tran-id: 092903CB0A7
*********************************************************************************
End of Request
```

The following is an explanation of the lock_trace output:

1. A shared physical lock was taken on page 11 of the iirelation table of the inv (inventory) database.

   Remember that physical locks are internal and are released as soon as possible.

2. A shared physical lock was taken on page 21 of the iiattribute table of the inventory database.
3. The lock on page 21 of the iiattribute table was released.
4. A shared physical lock was taken on page 19 of the iiattribute table of the inventory database.
5. The lock on page 19 of the iiattribute table was released.
6. The lock on page 11 of the iirelation table was released.
7. A shared physical lock was taken on page 11 of the iiindex table of the inventory database.
8. The lock on page 11 of the iiindex table was released.
9. An intended shared lock was taken on the parts table.

   This is the first lock in this example that was placed on a user table.

10. A shared lock was taken on page 0 of the parts table.
11. An intended exclusive lock was taken on the parts table.
12. An update lock was taken on page 0 of the parts table.
13. An exclusive lock was taken on page 0 of the parts table.
14. All locks used during this transaction were released.

Performance During Concurrency

When multiple users are performing selects, updates, inserts, and deletes on the same set of tables concurrently, consider the following when evaluating performance:

- If there are no users changing data in a set of tables, multiple, concurrent users reading data have no performance problems associated with concurrency. There are no deadlock problems, either.

  Once a writer mixes with the readers of a table, concurrent performance is affected, because the writer can acquire exclusive write locks on pages or tables. Deadlocks can occur, causing reduced performance for users who are "backed out" from the deadlock.
- Remember that locks acquired during a multiple query transaction are held until the COMMIT statement is executed. This can affect concurrent performance. Query-By-Forms uses multiple query transactions.
- Whenever possible, users must work in their own tables or download into their own tables with CREATE TABLE...AS SELECT statements. Doing so offloads tables where there is heavy concurrent activity.
- Nolock can be beneficial in certain situations.
- Use can be made of the Visual Forms Editor's form validations, rather than table-lookup validations that lock the reference table, because they are read only at form start-up time.

Approaches for Handling Heavy Concurrent Usage

In a heavy concurrent usage situation, there are two approaches:

- The "never-escalate-at-any-cost" approach

  Concurrent users are working in different regions of the table. Extreme care is taken by the person whose role it is to deal with concurrency problems (the system administrator or the DBA, or both), to ensure that nobody escalates to a table-level lock.
- The "table lock" approach

  This approach, which minimizes the occurrence of deadlock, applies when there is much concurrent activity on smaller tables or in one part of a larger table.

The Never Escalate Approach

The "never-escalate" approach is appropriate when the users are working in different parts of the table, they are running simple queries and updates, and making full use of primary and secondary indexes. The goal is to have users coexist as much as possible in the same tables, where no one impedes another user's performance by acquiring table locks.

Considerations of the "never escalate" approach include:

- A single-table keyed query starts with page locking, unless the SET LOCKMODE statement has been issued. Page locks are acquired until maxlocks is reached, at which point lock escalation occurs. By checking the tuple identifiers (tids) of rows visited, you can estimate the number of pages visited in a specific table.
- More complex queries can remove a table-level lock, if the query optimizer thinks that maxlocks pages are used.
- Make sure that you are using primary and secondary indexes effectively. Check how many pages are returned from a keyed, primary or secondary lookup to check that it is less than maxlocks for that table. The optimize database operation must be run at least on primary and secondary keys to help the optimizer make estimates.
- Monitor overflow levels in tables with ISAM and hash primary and secondary indexes.
- It is advisable to reduce fillfactors to lower than the default if tables with ISAM or hash storage structures are used, because this provides more room in the table after the modify.
- Make sure maxlocks is set to an appropriate figure, such as ten percent of table size.

When choosing storage structures while using the "never escalate" approach, the basic principle is that ISAM or hash structures with little or no overflow are better than small B-trees in a concurrent environment. The reason is that growing B-trees involve some locking when index pages split.

However, as the percentage of overflow builds up in the hash or ISAM structure, they become inferior to B-trees, because locks are held down overflow chains. In particular, if any overflow chain being visited is greater than maxlocks, escalation to table locks can occur. This increases the risk of deadlocks when there are multiple users in the same table.

At what point the trade-off occurs depends on the circumstances, such as how frequently MODIFY statements can be performed. Experimentation is advised. Overflow buildup must be checked in secondary indexes as well as primaries.

Concurrent performance analysis is much more difficult to analyze than single user performance. Be prepared to experiment using the guidelines presented.

The Table Lock Approach

The “table lock” approach is used only when there are unsolvable bottlenecks. The philosophy behind the approach says that it is better to have users queue up in an orderly manner to get into a table, thereby avoiding the risk of deadlock, than have them waste time backing out of deadlock situations.

Important! Before using this approach, ensure that lock escalation and transaction size are minimized.

This approach is appropriate when extensive table scanning is needed, as with set functions such as max and min. In these cases it is advisable to keep an extra table around containing max and min values, or to search for max and min values directly in a secondary index without reference to the base table.

In multiple query transactions, table-level locks reduce the likelihood of deadlocks but do not eliminate them. The following statement reduces the likelihood of deadlock in a multiple query transaction:

```
set lockmode on tablename
        where level = table;
```

This also applies to B-tree tables when they are small.

Under some circumstances setting READLOCK=EXCLUSIVE is useful. For example, when running a SELECT followed by an UPDATE statement.

Chapter 13: Performing Backup and Recovery

This section contains the following topics:

This chapter describes the following backup and recovery features of Ingres:

- Checkpointing and journaling to back up a database or selected tables
- Unloading a database
- Copying a database to back up particular tables or all objects you own in a database
- Using operating system backups to replace current or destroyed tables in a database
- Roll forward of a database to recover a database or selected tables from checkpoints and journals

The Need for Backup

You should back up your database regularly so that you can recover your data if necessary. Databases or tables can be damaged accidentally by hardware failure or human error. A disk crash, power failure or surge, operating system bugs, or system crashes, for example, can destroy or damage your database or tables in it.

Full or Partial Recovery

Ingres allows you to perform *full recovery*, which involves recovering an entire database, or *partial recovery,* which recovers selected tables in a database.

Partial recovery entails recovering data from a backup copy at a level of granularity finer than the entire database. In the event of failure, Ingres can, if possible, mark less than the whole database physically inconsistent. The advantage of partial recovery is that it reduces recovery times by requiring only recovery of logically or physically invalid data.

Logging System

The logging system keeps track of all database transactions automatically. It is comprised of the following facilities and processes:

- Logging facility, which includes the transaction log file
- Recovery process (dmfrcp)
- Archiver process (dmfacp)

Logging Facility

Each installation has an installation-wide transaction log file that keeps track of all transactions for all users. The log file can be distributed among up to sixteen partitions (locations), although Ingres treats the files as one logical file.

With dual logging enabled, the installation has an alternate log file. With dual logging, a media failure on one of the logs does not result in the loss of data or the interruption of service. If one of the log file disks fail, the logging system automatically switches to the other log without interrupting the application.

When log files are properly configured, the use of dual logging has a negligible impact on system performance.

Note: If your system is configured for Ingres Cluster Solution, each node in the Ingres cluster maintains a separate Archiver and Recovery error log. Each log is distinguished by having _*nodename* appended to the base log name, where nodename is the Ingres node name for the host machine as returned by iipmhost. Dual logging is also provided on clusters.

Log Space Reservation

During normal online processing, space is reserved in the transaction log file for possible use during recovery when it is rolling back transactions. The reserved space is used to write Compensation Log Records (CLRs), which describe the work performed during the rollback.

Generally, the logging system reserves approximately as much log file space to perform the rollback as was required to log the original operation. Exceptions are insert and update operations, which require less reserved space than the original log.

In the Log File page in the Performance Monitor window, you can see a close approximation of the log file space required for both normal log writes and for CLRs. Also displayed is the number of log file blocks reserved for use by the recovery system at any point in time. To access the Log File page, you click on the Log Information branch in the Performance Monitor window, and click the Log File tab in the Properties pane.

You can also accomplish these tasks using the sysmod command and the SET LOG_TRACE statement. For more information on the sysmod command, see the *Command Reference Guide*. For more information on the SET LOC_TRACE statement, see Set Log_Trace Statement (see page 397) and the *SQL Reference Guide*.

Recovery Process

The recovery process (dmfrcp) handles online recovery from server and system failures. The logging system writes consistency points into the transaction log file to ensure that all databases are consistent up to that mark and to allow online recovery to take place when a problem is detected. While a transaction is being rolled back, users can continue working in the database.

The recovery process is a multi-threaded server process, similar to a normal DBMS server. However, the recovery process does not support user connections. The process must remain active whenever the installation is active.

Archiver Process

The archiver process (dmfacp) removes completed transactions from the transaction log file and, for journaled tables, writes them to the corresponding journal files for the database. Each database has its own journal files, which contain a record of all the changes made to the database after the last checkpoint was taken. The archiver process “sleeps” until sufficient portions of the transaction log file are ready to be archived or until the last user exits from a database.

Data Verification Before Backup

As the DBA, you must know that the data in your database is good (can be accessed) before backing it up. Doing so can ensure that a successful recovery can be made if it becomes necessary to restore the database from the backup copy.

Methods of Verifying Data Accessibility

One method of verifying the accessibility of your tables is to write a script that automatically checks each of the tables and system catalogs in your database.

Otherwise, you can use one of the following suggested methods:

- Modify system tables to predetermined storage structures using the sysmod command.
- Modify user table storage structures using the modify command.
- Use any procedure that affects all the rows that are being backed up in each table. (For example, select all the rows from the tables.)

  If rows in a table are not accessible, you receive an error message. If this happens, restore the table from an earlier checkpoint before doing a new backup.
- Check the integrity of specific tables using the **verifydb -mreport -otable** *tablename* command.

For more information on these commands, see the *Command Reference Guide*.

Static or Dynamic Backup

You can make static (snapshot) backups of your entire database, or selected tables by using checkpoints.

To make a dynamic backup of your database, use checkpointing in combination with journaling.

These backup methods enable you to restore data up to the last checkpoint, or the last journaled transaction, respectively.

Backup by Checkpoints

Checkpoints provide you with a snapshot of the database at the time you took the checkpoint.

Each time you perform a checkpoint operation, a new checkpoint of the database is taken. Checkpointing can be performed at the database level and table level.

A record of up to 99 checkpoints can be maintained at any point. We recommend that at least one database-level checkpoint be included in this record.

We encourage that you use the infodb command to verify the status of the database and checkpoints. This ensures that a valid database checkpoint is always available.

Running a checkpoint (without any flags on the command) does not affect the current state of journaling for the database. For details on how to enable and disable journaling with a checkpoint, see Database Journaling (see page 364) and Disable Journaling (see page 366).

Tables that have had journaling enabled after the previous checkpoint have their journaling status changed from "enabled after next checkpoint" to just "enabled."

To checkpoint a database or tables, you must be a privileged user (operator privilege or system administrator).

Table-level Checkpoints

Generally, full database checkpoints are recommended over table-level checkpoints.

You should use table-level checkpoints only as a supplement to—not a substitute for—database-level checkpoints.

Note: Table-level checkpoints and recovery should be used cautiously. When using table-level checkpoints and restores, it is important—at the very least—to back up all dependent tables with a full checkpoint.

Recovery when using table-level checkpoints is restricted when the checkpointed table has been dropped or the table has been modified through any DDL statement. In these cases, the table-level checkpoint is rendered unusable. There is also danger in compromising the referential integrity of the database when rolling forward a table without journaling.

Performing table-level checkpoints on system catalogs is not permitted. We strongly encourage frequent database checkpointing of the iidbdb database.

Checkpoint a Database

A checkpoint is a snapshot of the database.

To checkpoint a database

Issue the following command at the operating system prompt:

```
ckpdb dbname
```

The ckpdb command creates a new checkpoint for the named database.

Checkpoint Selected Tables

Use table-level checkpoints only as a supplement to database-level checkpoints.

To checkpoint selected tables

Issue the following command at the operating system prompt:

```
ckpdb dbname [-table=tablename {, tablename}]
```

The ckpdb command takes a checkpoint of the selected tables in the database.

Checkpoint and Roll Forward of Tables

Whenever a database is rolled forward, we recommend that a new checkpoint be taken to allow subsequent table-level roll forward activities.

When a roll forward is performed at the table level, you can choose either to roll forward the table excluding or including all secondary indexes. You cannot specify a secondary index name as a table.

If it is necessary to do a roll forward with the No Secondary Index option, the base table's secondary index in the RDF cache become inconsistent. To clear the inconsistency, do one of the following:

- Drop or recreate the inconsistent secondary index
- Restart Ingres to refresh the RDF cache

If additional assistance is required, call Ingres Technical Support.

The Checkpoint Template File

A file called the checkpoint template file, cktmpl.def, drives the checkpoint and roll forward operations. The cktmpl.def file allows you to customize backup and recovery processes and provides additional information tracking. It is possible to modify the backup process so that the names of the tables that are specified during a table-level backup are written to a text file.

The II_CKTMPL_FILE environment variable overrides the default cktmpl.def file for a particular user. This override must be used when testing modifications to the cktmpl.def file before it is made available to the entire installation so that other users in the installation are not affected.

For checkpoint template codes and parameters, see Checkpoint Template File Description (see page 389).

Online and Offline Checkpoints

Checkpoints can be performed online or offline.

An *online* checkpoint, which is the default, can be performed while sessions are connected to the database. An online checkpoint stalls until any transactions running against the database are committed. Any new transactions started during the stall phase of the online checkpoint cannot run until the stall phase is completed.

An *offline* checkpoint can be performed when no one is using the database.

Perform an Offline Checkpoint

A checkpoint taken offline is performed when no one is using the database.

To perform an offline checkpoint

Issue the following command at the operating system prompt:

```
ckpdb -l dbname
```

The -l flag causes the checkpoint to be taken offline.

When using the -l flag, you can also use the "wait" flag (+w or -w):

+w

Waits for as long as necessary for the database to be free before taking the checkpoint.

-w

(Default) Returns an error message if the database is busy.

Checkpoints and Locking

By default, an exclusive lock is not taken on the database when you take a checkpoint. Other users who are using the database at the time of the checkpoint can continue working online. During this time, transactions in progress are placed in the dump file for the database.

When you perform a roll forward, the dump files are used to restore the database to its state when the checkpoint was taken. It updates the database from journals, if the database is journaled.

The following options on the ckpdb command, however, cause an exclusive database lock to be taken:

- -l to take the checkpoint offline
- +j or -j to enable or disable journaling

If you want to continue the present journaling status, use neither journaling option.

Delete Outdated Checkpoints

After you take a new checkpoint, you can delete previous checkpoints and journals.

To delete previous checkpoints and journals after you take a new checkpoint

Issue the following ckpdb command at the operating system prompt:

```
ckpdb -d dbname
```

Up to 98 checkpoints can be deleted.

Manual Deletion of Checkpoints

If you have taken more than 98 checkpoints since the last time you ran ckpdb -d, you must delete the additional old checkpoints manually using an operating system command.

Observe the following cautions when manually deleting checkpoints:

- Do this only *after* running ckpdb -d.
- Be sure that you do not delete the most recent checkpoint. You can identify the most recent checkpoint by its version number.

Checkpoint File Version Numbers

When you checkpoint a database, a checkpoint file is created for each location on which the database is stored. The names of the checkpoint files are in the format shown by the following example:

```
C000v00l.ckp
```

where *v* shows the version number of the checkpoint sequence and *l* shows the location number of the data directories. The most recent checkpoint file has the highest version number.

The latest version number is stored in the configuration file for the database. To determine this number, issue the following command at the operating system prompt:

```
infodb dbname
```

Delete Outdated Checkpoints Manually

If you have taken more than 98 checkpoints since the last time you ran ckpdb -d, you must delete the additional old checkpoints manually.

To delete old checkpoints manually

Use an operating system command, as follows:

Windows: Use the Windows del command from the II_CHECKPOINT\ingres\ckp\dbname directory.

UNIX: Use the UNIX rm command from the ii_checkpoint/ingres/ckp/dbname directory, where ii_checkpoint is the value of II_CHECKPOINT as displayed by the ingprenv command.

VMS: Use the VMS delete command.

Delete the Oldest Checkpoint

To delete the oldest checkpoint

Issue the format command at the operating system prompt:

```
alterdb dbname -delete_oldest_ckp
```

The oldest full database checkpoint, along with associated journal and dump files, are deleted.

Checkpoints and Destroyed Databases

Important! A checkpoint is a backup of an existing database. If you destroy the database (with the destroydb command), you cannot recreate it from a checkpoint, because this deletes a database's associated checkpoints as well.

To destroy your database and recreate it, use the unloaddb command. For more information, see the chapter "Loading and Unloading Databases."

Parallel Checkpointing in UNIX

In UNIX, you can checkpoint to a disk or a tape in parallel.

Checkpoint to Disk

To checkpoint a multi-location database to disk in parallel, issue the ckpdb command with the #m flag followed by the number of parallel checkpoints to be run. For example, to save two data locations at a time to the II_CHECKPOINT location, the command is as follows:

```
ckpdb \#m2 dbname
```

Checkpoint to Tape

To checkpoint a multi-location database to tape in parallel, in the Checkpoint dialog, specify multiple table devices to be used in the Tape Device edit control. For example, enter the following:

```
/dev/rmt/0m,/dev/rmt/1m
```

This saves one location per tape—the first location can be stored on device 0m; the second on device 1M. The third location can be stored on whichever device is finished first. The remaining locations can be stored on the next free device. The operator is prompted to insert a new tape for each location.

When performing parallel checkpointing to tape in UNIX, keep in mind the following:

- Recovery does not have to be in parallel if a checkpoint was done in parallel.
- Each tape label must include the checkpoint number, database name, and location number.
- Each tape device must be the same medium, that is, all 4mm or all 8mm; mixing is not permitted.
- The maximum number of devices that can be used is limited by the system's input and output bandwidth.

Putting Checkpoints on Tape in Windows

In Windows, the backup system uses the Windows backup utility to create checkpoints on tape. This utility allows you to back up on multiple tapes. The program prompts you for more tapes as needed during the checkpoint procedure.

The backup uses the commands in the following batch file:

```
%II_SYSTEM%\ingres\bin\ckcopyt.bat
```

You can tailor these commands to meet your needs (for example, to meet local conventions such as tape labeling).

For detailed information on backing up to tape, please see your Windows documentation on backup utilities.

Putting Checkpoints on Tape in UNIX

In UNIX, the backup system uses an operating system utility, such as tar (Berkeley UNIX) or cpio (System V), to create checkpoints. Both cpio and tar are limited to handling files that fit on a single tape. Because checkpoints of larger databases abort at the end of the first tape, you must estimate both the checkpoint size and the tape capacity before checkpointing these databases. If you estimate that the checkpoint exceeds the tape size, follow instructions in Checkpointing to Multiple Tapes in UNIX (see page 361).

How to Estimate Checkpoint File Size in UNIX

A separate checkpoint file is created for each location to which a database has been extended.

Follow these steps to estimate the size of checkpoint files:

1. Issue the following command at the operating system prompt:

   ```
   du ii_database/ingres/data/default/dbname
   ```

   where *ii_database* is the value of the environment variable II_DATABASE displayed by the ingprenv command.

   For other locations, substitute the name of the directory associated with the location name.

2. If your operating system uses tar, increase the resulting block size of the directory by 5%.

3. The du command displays the directory size in blocks. To get the file size in bytes, multiply the block size by the number of bytes in a block on your operating system.

For information on the number of bytes in a block on your system, see your operating system manual.

Tape Capacity in UNIX

The capacity of a tape depends on the following:

- Density at which the tape is written
- Length of the tape
- Size of the blocks written on the tape
- Length of the inter-record gap (IRG)

Standard 9-track tape drives write at either 800, 1600, or 6250 bits per inch (bpi), so the bits per inch specification is the same as bytes per inch. The standard tape length is 2400 feet.

Block sizes, which are not standardized, are important because of what is between the blocks—the IRG. A typical IRG is .75 inches of empty tape separating each block from the next.

Estimate Tape Capacity in UNIX

You can use the following formula to estimate the size of the file in bytes that a tape can accommodate:

$$F = (B + (I * D))/(12 * B * D * L)$$

where:

- F is the file size in bytes
- B is the block size in bytes
- D is the density in bits per inch
- L is the length of the tape in feet
- I is the IRG in inches

The sample file sizes in the following table were calculated for a standard 2400 foot tape, assuming an IRG of .75:

Tape Size	IRG	Block Size	Density	File Size (MB)
2400	.75	512	1600	13.8
2400	.75	512	6250	17.7
2400	.75	8192	1600	40.2
2400	.75	8192	6250	114.5

After using this formula to calculate the file size, you need to add an arbitrary amount to allow for miscalculations. You do not want a tape to run off the reel because you miscalculated the size of the file that fits. A reasonable amount to add is 5% of a tape’s capacity.

If your system uses a cartridge tape or other storage media, contact the vendor for the specifications that allow you to make the calculations described above.

Checkpointing to a Single Tape in UNIX

To checkpoint a database to a single tape:

1. Mount a tape reel.
2. In the Checkpoint dialog, enter the name of the tape drive in the Tape Device edit control.

   The equivalent ckpdb command at the operating system prompt is as follows with a tape drive named "/dev/rmt8":

   ```
   ckpdb -m/dev/rmt8 dbname
   ```

The backup created by this checkpoint writes over everything that was on the tape previously.

Checkpointing to Multiple Tapes in UNIX

When checkpoint files exceed the tape size, follow the appropriate procedure depending on whether the file fits on a disk.

When Checkpoint File Fits on a Disk

If the checkpoint file exceeds the size of the tape, but fits on a disk, follow these steps:

1. Follow normal procedures for checkpointing to disk.
2. Have your operating system administrator move the checkpoints from disk to tape. Use a standard system backup method, such as cpio or dump.

   If some of the database's tables are stored in alternate locations, separate checkpoint files are created for them in the checkpoint location. These files are small enough to be moves to single tapes.

   Caution! To System V Users: It is possible for large checkpoints to exceed the ulimit on your system. (The ulimit is a tunable operating system parameter that sets a limit on file size.)

When Checkpoint File Does Not Fit on a Disk

If the checkpoint file exceeds the size of the tape and does **not** fit on a disk, you must checkpoint the database using the operating system. To successfully checkpoint a database, you have to lock all users out during the entire process.

To lock out all users and take the checkpoint, follow this procedure:

1. To synchronize journaling, checkpoint the database to a null device by specifying the following options in the Checkpoint dialog:
   - Exclusive Lock
   - Wait
   - Delete Previous
   - Tape Device: /dev/null

   The Wait option causes the checkpointing to wait until all user locks have been released before beginning the checkpoint.

   The Delete Previous option removes all previous checkpoints and journals.

   The Tape Device specification causes the checkpoint to be placed in /dev/null, which is a nonexistent device. This makes the database "think" it is being checkpointed and causes journaling to be correctly synchronized. At this time, all changes to the database are guaranteed to be on disk.
2. To lock the database, start a new process:

   C shell:

   After the first message from the checkpoint is printed, press Ctrl+Z.

   Bourne shell:

   Log in at another terminal immediately after the checkpoint begins.

   Start the new process by issuing the following command at the operating system prompt:

   ```
   ingres -l +w dbname
   ```

   The +w flag causes a wait until that lock is granted.
3. After the checkpoint finishes:

   C shell:

   If the checkpoint process is stopped (csh job control), put the job back in the foreground; wait for the process to complete.

   Bourne shell:

   Wait for the process to complete.
4. Have your operating system administrator use standard system backup methods to back up the database directory to tape.

Make sure that the backup method used allows you to save the files and recover them to their original places on the system. Some backup methods have limitations. The volcopy command, for instance, requires that the database disk device be unmounted and unavailable for use by any users during the copy. Additionally, it saves files by saving the entire file system.

5. For the C shell:

   Leave the second process stopped (csh).

   For the Bourne shell:

   Leave the second process at the SQL prompt (*) until the backup is complete.

6. Quit from the SQL prompt held by the second process.

Putting Checkpoints on Tape in VMS

To initiate a checkpoint in VMS, ready the tape and issue the ckpdb command with the –m option. For more information about the ckpdb command, see the *Command Reference Guide*.

The backup system uses the VMS BACKUP utility to create checkpoints. This utility allows you to back up on multiple tapes. The program asks for more tapes as needed during the checkpoint procedure.

The backup uses the following command in the script:

```
II_SYSTEM:[INGRES.FILES.CHECKPOINT]CKP_TO_TAPE.COM
```

You can tailor this command to meet your needs (for example, to meet local conventions such as tape labeling).

For detailed information on backing up to tape, see your VMS backup documentation.

Journals

For a dynamic backup of your database, use journals in combination with checkpoints.

While checkpoints provide you with a snapshot of the database, journals keep track of all changes made to journaled tables after the last checkpoint.

When you are journaling a database, you should do the following:

- Take regular checkpoints of your database to minimize recovery time.
- Periodically verify that your journaling data is correct by auditing the database. For information, see Audit Trails (see page 373).

Tools for Performing Journaling

You can perform journaling tasks using system commands or in VDBA.

The system commands for journaling tasks are the ckpdb and alterdb commands. For more information, see the *Command Reference Guide*.

For the detailed steps for performing journaling procedures in VDBA, see the Procedures section of online help.

Database or Table-level Journaling

Journaling can be selected for an entire database or on a table-by-table basis.

Database Journaling

The recommended approach is to journal the entire database rather than specific tables. Tables in journaled databases are created "with journaling" if that is the default_journaling setting of the server class used by the Ingres DBMS Server you are connected to.

Disable journaling on specific tables only if a rollforward recovery of those tables is not important. You must exercise caution when creating non-journaled tables in journaled databases. Non-journaled tables cannot be audited when the database is audited, in addition to their lack of roll forward recovery. Following a roll forward recovery, the relationship between journaled and non-journaled tables can be confusing.

Table-level Journaling

If you choose to journal selected tables, you are responsible for ensuring that all related objects are also journaled (for example, that all tables associated with a view are journaled).

Enable Journaling on an Entire Database

To journal an entire database

Issue the following command at the operating system prompt:

```
ckpdb +j dbname
```

or

Use the set journaling (or ING_SET "set journaling" equivalent) statement to enable journaling of all activities associated with the database.

Note: The only tables that are enabled are those whose journaling status is “enabled after next checkpoint.” Tables whose journaling status is “disabled” cannot be enabled.

New Tables and Journaling

The journaling of new tables begins, as follows:

- If you have enabled journaling on the database and the table is created with journaling enabled, the new tables begin journaling *immediately*.
- If you have not enabled journaling on the database, the new tables begin journaling after you take a checkpoint with the enable journaling option (although tables created with journaling disabled are never enabled even after journaling is enabled for the database as a whole).

Start Journaling on a Database Not Checkpointed

To start journaling on a database that has not yet been checkpointed

Issue this command:

```
ckpdb +j dbname
```

Journaling and Online/Offline Checkpoints

An explicit journaling option on the ckpdb command causes the checkpoint to be taken offline and with an exclusive lock on the database.

The first time journaling is turned on in a particular database, you *must* checkpoint the database with journaling enabled (ckpdb +j *dbname*). Doing so ensures that the checkpoint is taken offline.

Once you have enabled journaling by checkpointing offline with the +j option, you can maintain the "journaling on" status and take *online* checkpoints by not subsequently setting the +j option when you take a checkpoint. Online checkpoints permit users to continue using the database while the checkpoint is being taken.

After you have enabled journaling for the database by checkpointing offline with the +j option, you can take an offline checkpoint to start journaling of tables for which journaling is enabled after the next checkpoint.

Disable Journaling

To disable journaling

Use either of these methods:

- With the WITH NOJOURNALING option of the CREATE TABLE statement on new tables.

  For example, to turn journaling off when you create the emp table, issue this statement:

  ```
  CREATE TABLE emp
  name varchar(20),
  age i2,
  salary money)
  WITH NOJOURNALING;
  ```

- By setting journaling off for an entire session with the SET NOJOURNALING option of the SET statement, for example:

  ```
  SET NOJOURNALING;
  ```

Stop Journaling on a Table

To stop journaling a particular table

Issue the following statement from the query language monitor:

```
SET NOJOURNALING ON tablename;
```

Methods for Stopping Journaling on All Tables

You can stop journaling all the tables in a database with either of the following commands:

- Altering a database using the alterdb command.
- Creating a checkpoint using the **ckpdb -j** command.

  Note: This takes effect immediately; therefore, it must be used only for emergencies. For information, see Disabling Journaling When Checkpointing (see page 367).

To re-enable journaling on a table or database that has had journaling disabled, use the ckpdb command, as described previously.

Disable Journaling When Checkpointing

The following command issued at the operating system prompt stops journaling of all the tables in a database:

```
ckpdb -j dbname
```

A checkpoint of the specified database is taken, and then journaling is stopped. After journaling is stopped, you can still take periodic checkpoints of the database.

Disable Journaling When Altering a Database

When you disable journaling using the alterdb command, journaling of a database is halted immediately, regardless of whether users are connected to the database.

This option is provided as a method for recovering from journaling system problems that prevent the archiver from moving transaction log file records to the database journal files, for example, if the disk partition containing the journal files is not periodically purged of obsolete journal files and the partition becomes full. If the logging system is unable to move records from the log file to the journal files, the transaction log file eventually fills up, causing a LOGFULL condition. When this occurs, no database activity can proceed until the LOGFULL state is cleared.

Important! Using this option to disable journaling makes the displayed value for the journaling status inconsistent. Tables are "journaling enabled," even though journaling is disabled for the database as a whole and you expect to see "enabled after next checkpoint."

To use alterdb to disable journaling

The following procedure must be run by the DBA of the database. It does not require a database lock and can be run even while the log file is full (LOGFULL).

1. To disable journaling on a database, issue the following command at the operating system prompt:

   ```
   alterdb dbname -disable_journaling
   ```

   The database is no longer journaled.

   Caution! Do not use rollforwarddb on a database that has journaling disabled. Any transactions committed after the alterdb action, or that were still in the transaction log file at the time journaling was disabled, will be lost.

2. To check the database state, use the infodb command at the operating system prompt:

   ```
   infodb dbname
   ```

   The infodb listing will indicate whether journaling has been disabled.

3. To restart archive processing after disabling journaling on the database, issue the following command at the operating system prompt:

   ```
   ingstart -dmfacp
   ```

4. Schedule a new checkpoint to re-enable journaling as soon as possible, by using the following command at the operating system prompt:

   ```
   ckpdb +j dbname
   ```

Database Characteristics Affected by Alterdb

The alterdb command lets you disable journaling and change several database characteristics, including:

- Change journal block settings
- Delete oldest checkpoint
- Set verbose mode

To perform this operation, you must be the owner of the database or have the operator privilege.

Journal File Size

Journal files are created by the archiver process by the first journal write after a checkpoint takes place. Additional journal files are created as prior files are filled.

By default, journal files are created with:

- A target number of journal blocks of 512
- A block size of 16, 384 bytes
- An initial allocation of 4 blocks

This results in a target journal file size of 8 MB (16, 384 * 512 bytes). Although most users find these parameters satisfactory, all three can be modified by using the alterdb utility.

The alterdb command has the following syntax for altering block sizes:

```
alterdb dbname -target_jnl_blocks=n | -jnl_block_size=n | -init_jnl_blocks=n
```

Target Journal Size

The alterdb command specifies the target journal size in the the following format:

```
alterdb dbname -target_jnl_blocks=n
```

where *n* is the number of blocks between 32 to 65536.

A journal file is closed and a new one is created when either a checkpoint is taken (actually, when the first write after a checkpoint is taken) or when the journal file fills.

The -target_jnl_blocks=*n* option of alterdb allows some control over when the logging system declares a journal file full. This parameter is known as the “target journal file size” because the exact size of a journal file cannot be easily predicted. The archiver closes off journal files, if they grow larger than the target number of blocks, only at the completion of an archive cycle. Longer archive cycles imply more variation in journal file sizes.

Upon successful completion of this command, a message is written to the errlog.log. The updated block value can be observed as the infodb parameter “Target journal size”.

The command takes effect immediately (or more accurately, the next time the archiver reads the configuration file).

The initial journal size (init_jnl_blocks) may be affected by this command.

Journal Block Size

The alterdb command specifies the journal block size in the format:

```
alterdb dbname -jnl_block_size=n
```

Only one database name is required. Valid journal block sizes are 4096, 8192, 16384, 32768, and 65536 bytes.

Archiver (dmfacp) performance is affected by the journal file block size. You normally change the block size (Size edit control) in conjunction with the number of target journal blocks (-target_jnl_blocks). Doing so allows you to target the creation of journal files of a given size. Changing the block size without also changing the number of blocks in a journal file changes the target size of the file.

You typically change the journal block size immediately after the database is created, before the initial checkpoint is taken with the journaling option. Thereafter, changing the journal block size is generally required only for installations with a relatively high volume of journaled data. You can only change the journal block size when journaling is *not* currently enabled.

To change the journal block size on a database that is currently journaled

1. Take a checkpoint and disable journaling:

   ```
   ckpdb -j dbname
   ```

2. Set the journal block size:

   ```
   alterdb dbname -jnl_block_size=n
   ```

3. Take a checkpoint and enable journaling:

   ```
   ckpdb +j dbname
   ```

   When this operation completes successfully, a message is written to the errlog.log. The updated journal file block size can be observed as the infodb “Journal block size” parameter.

Initial Journal Size

The alterdb command specifies the initial journal size in the format:

```
alterdb dbname -init_jnl_blocks=n
```

where n is a number of blocks from 0 to the current target journal size (which can be obtained using infodb). Only one database name is required. The -init_jnl_blocks=*n* option allows a measure of control over when journal file disk space allocation takes place, but only for the first journal file created after a ckpdb command.

This alterdb command can be issued at any time, and takes effect when the next database journal file is created. In the case of an offline checkpoint, this can be some time after the ckpdb command is issued. In the case of an online checkpoint, the file allocation occurs during execution of the checkpoint.

Upon successful completion of this command, a message is written to the errlog.log. The updated block value can be observed as the infodb "Initial journal size."

Considerations When Resizing Journal Files

Preallocating space in journal files using alterdb can reduce the likelihood of running out of journal file disk space.

Filling a journal file causes the archiver to stop, and if left untreated, eventually causes the log file to fill, which brings the system to a halt.

With the alter database operation you can, for example, request creation of journal files of a given size and also request preallocation of the entire file. If the file is sufficiently large, this eliminates the possibility of running out of journal disk space during normal online processing.

This can, however, cause unused journal space to be wasted. If excessive space is allocated during journal file creation, that disk space can be made unavailable when a subsequent checkpoint operation takes place.

If it is necessary to control journal file size more accurately, the archiver must be awakened more frequently. This can be accomplished with smaller consistency point (CP) intervals, allowing more frequent archiver "wake-ups." The consistency point interval can be configured using CBF (or the Configuration Manager, if available). Smaller CP intervals can affect system performance, although the processing involved is for a short interval of time.

Considerations When Resizing Journal Files on UNIX

On UNIX systems, disk space must be physically written when a journal file is extended. When a journal file is filled, a new one is created. It is undesirable for performance to be affected by file allocation that occurs at unplanned intervals.

You can use the alter database space preallocation features to manage when the allocation takes place, allowing control over when the allocation time delay occurs. A significant amount of journal file I/O can occur when the first journal file is created, with the archiver being unavailable during this time. This can be observed as an online checkpoint taking a long time to complete, or the archiver performing a large amount of work when the first journal write after an offline checkpoint takes place.

Audit Trails with Journals

In addition to using journals for recovery, you can use journals to produce audit trails of changes to a database. You must be the DBA for the database or have the security privilege to perform an audit on a database.

Audit your database periodically to verify that your journals are correct.

Tools for Auditing a Database

The audit database operation is performed using the auditdb command. For complete details, see the *Command Reference Guide*.

In VDBA, this operation is performed using the Audit Database dialog, invoked by the Operations Audit menu command. For the detailed steps for performing this procedure, see the Procedures section of online help.

Understanding the Audit Operation

The auditdb command lets you produce a listing or file of changes made to journaled tables after the last checkpoint. This listing may not include *all* changes that have been made after the last checkpoint for the following reasons:

- Because auditdb does not exclusively lock the database, other users can complete a transaction while the audit is running.
- If other users are using the database when you perform an audit, a completed transaction may not have been moved to the journal files.

The audit database operation scans journal files twice. A prescan is performed to filter out undesired information (for example, aborted transaction data). The second scan outputs journal records of interest. To improve program performance, the -e option (Before edit control value in VDBA) terminates both scans when an End Transaction record is found that has a time later than that specified.

The -inconsistent option lets you view journals that the database has marked as inconsistent. **Note:** The audit database operation can still fail if core catalogs are inconsistent.

The -wait option makes the audit wait until journals are current. "Current" in this context means either of the following:

- No further archiving is required on the database.
- The archiver has copied all log file information up to the log file end-of-file when the audit database request was initiated.

Note: If a large amount of unarchived information remains in the log file when this request is initiated, a significant delay in processing can occur.

How to Load an Audit Trail as a Table

To make querying the data easier, you can create an audit trail as a file in your current directory and load the file into a table in your database.

To do this, follow these steps:

1. When you create the audit trail, use the -file flag to create an audit trail file in the current directory.

   Note: You must have first specified at least one table. Also, you can specify files only if the table you are auditing has fewer than 1940 bytes per row.

   In the following example, auditdb extracts a record of the changes to the employee table from the journal for the demodb database. It places the changes in the current directory in a file named empaudit.trl.

   ```
   auditdb -table=employee -file=empaudit.trl demodb
   ```

2. To copy the file into a database table, create a table to hold the audit trail data.

   When creating the table, include the audit trail and employee table columns shown below. Enter the audit trail columns before the table's columns, in the order shown. If you do not, the copy operation can fail when you try to copy the audit trail data into the table.

Column Name	Data Type	Description
date	date not null with default	Date and time of the beginning of the multi-query transaction that contained the operation
username	char(32) not null with default	User name of the user who performed the operation
operation	char(8) not null with default	Insert, update, or delete operation
tranid1	integer not null with default	Transaction identification number. Concatenated with tranid2.
tranid2	integer not null with default	Transaction identification number. Concatenated with tranid1.
table_id1	integer not null with default	Table identification number. Corresponds to value in table_reltid column of iitables system catalog for specified table.

Column Name	**Data Type**	**Description**
table_id2	integer not null with default	Table identification number. Corresponds to value in table_reltidx column of iitables system catalog for specified table.
name	varchar(20)	Employee name
age	integer	Employee age
salary	money	Employee salary
dname	varchar(10)	Department name
manager	varchar(20)	Employee manager

In the following example, a table named empaudit is created to hold the data from the empaudit.trl file:

```
create table empaudit
(date date not null with default,
username char(32) not null with default,
operation char(8) not null with default,
tranid1 integer not null with default,
tranid2 integer not null with default,
table_id1 integer not null with default,
table_id2 integer not null with default,
name varchar(20),
age integer,
salary money,
dname varchar(10),
manager varchar(20));
```

The last five columns are from the employee table.

3. Use the COPY statement to load the new table with the data from the file from Step 2.

   In the following example, the data in the empaudit.trl file is copied to the empaudit table:

   Windows:

   ```
   copy empaudit() from 'C:\users\joe\empaudit.trl';
   ```

   UNIX:

   ```
   copy empaudit() from '/usr/joe/empaudit.trl';
   ```

   VMS:

   ```
   copy empaudit() from '[usr.joe]empaudit.trl';
   ```

   The table created from the audit trail (in this example, the empaudit table) contains:

   - A row for each row added to the employee table
   - A row for each row removed
   - Two rows for each update: one showing the row before the update and the other showing the row after the update

Backup by Copying

You can copy a database (using the copydb command) to back up the tables, views, and procedures that *you own* in a database.

Because any user authorized to use a database can use the copy database operation, this is a useful backup method for a non-DBA, who can use it to back up tables, views, and procedures.

By default, all of the tables, views, and procedures that you own in the database are copied. If you specify table names, only those tables are copied.

For a complete explanation of the copy database operation, see the chapter "Loading and Unloading Databases."

Back Up Tables with Copydb Command

Before using this procedure, see the "Loading and Unloading the Database" to understand how copydb works.

To back up tables with copydb

1. Create a temporary working directory for the copy.in and copy.out scripts and move to this directory. For example, you might issue the following commands at the operating system prompt:

   Windows:

   ```
   mkdir D:\tmp\mydir.backup
   D:
   cd \tmp\mydir.backup
   ```

   UNIX:

   ```
   mkdir /tmp/mydir.backup
   cd /tmp/mydir.backup
   ```

   VMS:

   ```
   create/dir MYDIR.BACKUP
   set default [MYDIR.BACKUP]
   ```

2. To back up specified tables, issue the following command at the operating system prompt:

   `copydb` *`dbname tablename`* `{`*`tablename`*`}`

   To back up all the tables, views, and procedures that you own in the database, issue the following command at the operating system prompt:

   `copydb` *`dbname`*

   This creates copy.out and copy.in scripts for the objects copied.

3. To copy the data out of the database, issue the following command from the operating system:

   `sql` *`dbname`* `<copy.out`

   This creates a copy of the objects copied from your database.You can store these files on tape or leave them on disk.

To restore data from a copydb backup, you run the copy.in script.

Backup by Unloading

Unloading a database is a time-consuming method for backing up and recovering your database, because all of your database's files must be unloaded and reloaded. For this reason, we recommend that you use checkpointing instead.

Unloading a database, however, can be useful as a backup tool because it enables you to:

- Generate copy scripts, which can be used to recreate your database.
- Recover particular tables by editing the copy.in scripts. For a description of the copy.in scripts, see the chapter "Loading and Unloading Databases."

To accomplish this task using a system command, use the unloaddb command. For more information, see the *Command Reference Guide*.

For the detailed steps for generating these scripts using VDBA, see the Procedures section of online help. See the Creating Unload and Reload Scripts topic.

Recovery

To recover a database from checkpoints and journals or from checkpoints only, you use the *roll forward* operation. This operation lets you recover the following:

- A non-journaled database from a checkpoint
- A journaled database from checkpoints and journals
- A database from a tape checkpoint
- Selected tables

Rollforward Operation

Performing a roll forward of a database overwrites the current contents of the database being recovered.

To perform a roll forward, you must be the DBA for the database or have the operator privilege.

When you roll forward a database, the database is locked to prevent errors from occurring. If the database is busy, the roll forward operation waits for the database to be free before recovering it. (If you specify the wait [+w] option, the rollforwarddb operation pauses until all users have left the database. If you do not specify the wait option, you get a message that the database is in use.)

If the target checkpoint was taken online (when the database was in use), the roll forward operation does the following:

- Restores the database from the checkpoint location to the database location.
- Applies the log records in the dump location to the database, which restores the database to the state when the checkpoint began. The log records contain the transactions that were in progress when the checkpoint was taken.

  This step is not performed when restoring a database from an offline checkpoint because there were no transactions in progress during an offline checkpoint.
- Applies the journal records to the database, if the database is journaled.

Note: A roll forward can write Compensation Log Records (CLRs) to the transaction log file while executing the rollback phase of a roll forward recovery. This happens rarely, only if incomplete transaction histories are written to the journals. This is an unlikely condition except when the transaction log file is lost (or, if running with dual logging, when both copies are lost). In this case, it is possible for journal files to grow in size as a consequence of performing a roll forward.

Tools for Performing a Roll Forward Operation

The system command to roll forward a database is the rollforwarddb command. For details on this command, see the *Command Reference Guide*.

In VDBA, to roll forward a database, use the Roll Forward DB dialog, invoked by the Database Rollforward DB menu command. For the detailed steps for performing this procedure, see the Procedures section of online help for VDBA. See the Recovering a Database from Checkpoints topic.

Recover a Journaled Database

To recover a specific database from the last checkpoint and journal, where both the checkpoints and journals are stored online, issue the following command at the operating system prompt:

```
rollforwarddb dbname
```

Note: All journals since the last checkpoint must be present.

Apply Journals Incrementally to a Backup Database

As journal files are generated, you can apply them incrementally to a backup copy of the database. Doing so minimizes downtime if the backup database is needed for disaster recovery.

To apply journals incrementally

1. Start the incremental rollforwarddb by issuing the following command:

   ```
   rollforwarddb dbname +c -j -incremental
   ```

2. Discover and apply new journals by issuing the following command:

   ```
   rollforwarddb dbname -c +j -incremental -norollback
   ```

   The database remains inconsistent and readonly. There may be open transactions.

3. Discover and apply new journals and roll back open transactions by issuing the following command:

   ```
   rollforwarddb dbname -c +j -incremental -rollback
   ```

   The -rollback flag ends the incremental rollforwarddb, and the database is marked consistent and updatable.

Note: Incremental rollforwarddb requires that all journals since the last checkpoint be present. For example, if you apply a batch of journal files, and then delete the previous batch of journal files, **rollforwarddb -incremental -rollback** may fail.

For details on the -incremental and other flags, see the rollforwarddb command description in the *Command Reference Guide*.

Recover a Non-Journaled Database

To recover a non-journaled database from the last checkpoint, issue the following command from the operating system prompt:

```
rollforwarddb +c dbname
```

Recover a Database from Tape Checkpoints

To recover a database whose checkpoints are on tape

1. Mount the tape reel containing the checkpoints.
2. Issue a rollforwarddb command at the operating system prompt, naming the tape drive as the device:

   ```
   rollforwarddb +c [+j] -mdevice dbname
   ```

   The checkpoint is read from the tape and the journal files are applied, if the database is journaled, to bring your database up to date.

Parallel Roll Forward from Disk (UNIX)

To roll forward a multi-location database to disk in parallel, issue the rollforwarddb command the #m flag followed by the number of parallel restores to be run.

For example, to restore two data locations at a time from the II_CHECKPOINT location, the command is as follows:

```
rollforwarddb #m2 dbname
```

Parallel Roll Forward from Tape (UNIX)

To roll forward a multi-location database from tape in parallel, specify the devices to be used in the From Tape Device edit control. For example, the following tape device can be specified:

```
/dev/rmt/0m,/dev/rmt/1m
```

This restores one location per tape—the first location can be restored from device 0m; the second location can be restored from device 1M. The third location can be restored from whichever device is finished first. The remaining locations can be restored from the next free device. The operator is prompted to insert the numbered tape into the free device.

Some points to be aware of when performing parallel roll forward from tape in UNIX include:

- Recovery does not have to be in parallel if a checkpoint was done in parallel.
- Recovery can be in parallel if a checkpoint was not done in parallel.
- Each tape label must include the checkpoint number, database name, and location number.
- Each tape device must be the same medium, that is, all 4mm or all 8mm; mixing is not permitted.
- The maximum number of devices that can be used is limited by the system's input and output bandwidth.

Table Recovery Using Roll Forward

You can specify that only certain tables are recovered during a roll forward database operation. (Journals of tables in the database must be enabled.) When doing table-level recovery, you can optionally move the table to a new location.

Note: The database must be extended to the new locations before the rollforward.

The format for recovering tables is as follows:

```
rollforwarddb dbname[/server_class]
  [-table=tablename {, tablename}
  [-nosecondary_index] [-on_error_continue]
    [-relocate -location=locationname {, locationname}
     -new_location=locationname {, locationname}]]
```

Note: Table recovery is not allowed if structural changes have been made to the table after the checkpoint (that is, if you have modified the table, created indexes or altered the number of columns in the table).

Retract Changes Using Roll Forward

If a user makes a serious error in a table that is being journaled, the changes can be retracted. Use the roll forward operation to restore the database up to the beginning of the transaction in which the error occurred.

For example, to restore a database from the previous checkpoint to its condition at 8:00 A.M. on August 15, 2008, issue the following command:

```
rollforwarddb -v +c +j -e15-apr-2008:08:00:00 dbname
```

This command retracts *all* changes made to the database after this time, not just those made to the table with the error.

To ensure that the error is not reintroduced when you perform a roll forward in the future, take a new checkpoint to reset the journals.

Recover a Subset of Data Using Roll Forward

The roll forward end time option (specified with the -e flag) permits the recovery of a subset of data in the journal file. The option is useful when problems have been encountered in a full roll forward database operation or when, for example, a critical piece of data has been inadvertently deleted.

Important! As this form of recovery does not restore the database to the state reflected by the full set of journals, it is critical that a checkpoint of the database be performed after the recovery completes. If not, another roll forward performed later can leave the database in an inconsistent state.

The only recommended course of action after is rolling forward a database with the -e option is:

- Roll forward the database again
- Checkpoint the database, preferably with the -d option (to delete previous checkpoints)

Note: The rollforwarddb -b and -e (Before and End) options operate on End Transaction timestamps, not on the time that a user may associate with an update. The auditdb -e and -b (End and Before) options also operate on End Transaction timestamps, and can be used to check anticipated roll forward results.

Recover a Database from an Old Checkpoint

If the most recent checkpoint has been damaged or is unreadable, it is possible to recover from an older checkpoint. You can use either a specific checkpoint number or the most recent usable checkpoint.

To recover the database from a particular checkpoint and apply all journals after that time, issue the following command:

Windows:

```
rollforwarddb +j #cn dbname
```

UNIX:

```
rollforwarddb +j '#cn' dbname
```

VMS:

```
rollforwarddb +j #cn dbname
```

where *n* is the checkpoint number. For example the following command requests recovery from checkpoint 4 for the Employee database:

```
rollforwarddb +j #c4 employee
```

The checkpoint sequence number must be a valid checkpoint number. You can verify this number with the infodb command.

If the most recent checkpoint is unfinished and you want to recover using the most recent usable finished checkpoint, issue the following command:

Windows:

```
rollforwarddb +j #c dbname
```

UNIX:

```
rollforwarddb +j '#c' dbname
```

VMS:

```
rollforwarddb +j #c dbname
```

The #c flag can also be used with the -b and -e flags s if you want to restore a database to its state at some previous moment in time.

Caution! You must exercise extreme caution with the -b and -e options. Because these commands roll the database forward to a point in time other than that fully represented by the journals, transactions that were performed after the -e time or before the -b time are lost. Partially completed transactions can be backed out by the roll forward process. Furthermore, a checkpoint must always be performed after completion of such a roll forward, thereby ensuring that obsolete journal data is not inadvertently reused in a subsequent recovery (or by an audit database operation to produce inaccurate auditing results).

Note: The audit database -b and -e flags behave in the same manner as the equivalent roll forward flags, and can be used to predict roll forward results.

Recover from the Loss of the Transaction Log File

In the unlikely event of a loss of the transaction log file (or, if dual logging is enabled, loss of both file copies), the following recovery procedure can be used to restore as much database information as is possible. Follow these steps:

1. Create a new transaction log file. For more information, see the *Installation Guide*.
2. The next action differs, depending on whether offline or online backups take place. Included in the latter class of systems are those that employ journaling capabilities.
   - For offline backups

     Installations using their own backup and recovery mechanisms (implying no use of online checkpoint or journaling facilities) only need to restore database directories and bring the system back up. No directed recovery is needed, after backups are done during a period when there is no system activity, and when all database information is resident on disk.
   - For online backups and roll forward

     If you are using online checkpoints and journaled databases, bring the installation back up with the newly initialized log file. All databases open at the time of the failure can be marked inconsistent by the recovery process. Each must be recovered in turn by the roll forward database operation. The +j (Enable Journaling) option with roll forward is specified for journaled databases; this option is not specified for those databases that are not journaled.

Note: A roll forward operation restores databases to a consistent state even if incomplete transaction histories have been copied to the journal files.

Checkpoint Template File Description

The checkpoint template file drives the checkpoint and roll forward operations. If needed, you can tailor the file to meet the requirements of your site.

For example, if the database exists on multiple locations, checkpointing backs up each location to a separate tape or disk and, in turn, roll forward restores each location one at a time. If you want to use a different backup method or only one tape for all locations, you can edit this command file.

Checkpoint Template Codes

In the checkpoint template file, a four-character uppercase code at the beginning of each line provides the following information:

The first character indicates when the command is to be used. Valid characters are:

B (Begin)—the command is to be executed before the device is used. It indicates setup work done prior to the execution of the command.

P (Prework)—the command is to be executed before the work is executed.

I—the command begins table-level recovery (initializes only).

W (Work)—the command activates the device. It indicates the execution of the command.

F—the command ends table-level recovery (comments only).

E (End)—the command is executed after the device is used. It indicates cleanup work done after the operation is complete.

The second character indicates whether the command specifies several types of checkpointing and roll forward options. Valid characters are:

S—the command is for checkpointing only.

R—the command is for roll forward only.

E—the command is for both checkpointing and roll forward.

D—the command is for delete file processing.

C—the command checks if a database checkpoint exists before the roll forward.

J—journals are to be applied, for a roll forward.

U—dumps are to be applied, for a roll forward.

The third character specifies the device. Valid characters are:

T—the command on that line refers to reading from or writing to a tape.

D—the command refers to disk operations.

E—the command applies to both types of devices.

The fourth character specifies the data. Valid characters are:

D—the command is for a database.

A—the command is for all databases.

T—the command is for table(s).

E—the command is for either a database or table.

R—the command is for a raw location (database and table level are the same)

Examples: Checkpoint Template Code

Here are examples of a checkpoint template code:

WSTD identifies the command line to use during the working (W) phase of a checkpoint which is saving (S) a database to tape (T), for a database (D).

BRDT identifies the command line to use during the begin (B) phase of a roll forward operation that is restoring (R) from disk (D) for a table (T).

Substitution Parameters

The checkpoint template file can optionally include substitution parameters that can be filled in at run time, to specify things like:

- Which database directory to back up
- Which tape device the user specified in the Checkpoint dialog

The parameters consist of a "%" and a single uppercase character, as follows:

%T

The type of operation: 0 if to tape, 1 if to disk.

%N

The total number of locations being written.

%M

For the begin or end operations, the incremental/current location number. For save or restore operations, this starts at 1 and is incremented after each save or restore command.

%D

The path to the database directory being saved or restored.

%C

The path to the checkpoint directory of disk files or the device name if to tape.

%F

The name of the checkpoint file created or read from.

%A

%C prepended to %F in a form to produce a fully specified file (that is, %A = %C/%F).

%X

The name of the table, pertinent to the work commands executed under table processing.

%B

Expanded during execution to represent the list of internal files that are associated with a table checkpoint. This parameter is pertinent to the work commands executed under table processing.

The "%" parameters in the commands are replaced by ckpdb and/or rollforwarddb when the command is executed.

Valid Code Combinations in the Checkpoint Template File

The valid code combinations in the checkpoint template file are shown here:

B	[S,R,E,J,U] [T,D,E] [T,D,E,A]
P	[S,R] [T,D] [D,T]
W	[S,R,E,J,U,D,C] [T,D,E] [T,D,E,A]
I	[,R,E] [T,D,E] [T,E]
F	[,R,E] [T,D,E] [T,E]
E	[S,E] [T,D] [D,T,E]

For every entry with a first character of B, there must be an accompanying entry beginning with E.

This section demonstrates how the codes are used in the checkpoint template file to perform checkpointing and roll forward operations in a variety of ways.

Checkpointing

The checkpointing operation (ckpdb command) executes the following sequence of codes in the cktmpl.def file:

Bs*xy*	Beginning checkpoint
Ws*xy*	Executed once for each location
Es*xy*	Ending checkpoint

where:

x denotes D for disk, T for tape, or E for both.
y denotes D for database, T for table, or E for both.

Roll Forward

The roll forward operation (rollforwarddb command) processes the following codes in the cktmpl.file:

WC*x*A for each location

If table processing is specified, the following codes are executed:

BR*x*T	once per location
IR*x*T	once per location
WD*x*T	for each table
WR*x*T	for each table
FR*x*T	once per location
EE*x*E	once (note that ER*x*T is executed if available)

If an entire database is being recovered (rather than specific tables), the following codes are executed:

BR*x*D	once for each location
WD*x*D	once for each location
WR*x*D	once for each location
EE*x*E	once (note that ERxD is executed if available)

For all roll forward operations, the following codes are executed:

BU*x*A	if dumps are to be applied
WU*x*A	
EE*x*E	
BJ*x*A	if journals are to be applied
WJ*x*A	
EE*x*E	

Format of the Checkpoint Template File in Windows

The checkpoint template file uses the two batch files, ckcopyd.bat (for checkpointing to disk) and ckcopyt.bat (for checkpointing to tape).

The checkpoint template file, cktmpl.def, can be found in the folder %II_SYSTEM%\ingres\files.

Each line contains a command preceded by a four-character code that tells when to use the command.

By altering this file, or the two batch files that it calls, you can change how checkpoints are performed. You can add or delete flags to the underlying operating system commands, or you can supply your own batch files to perform the backup and restore steps.

For example, the command:

```
BSTD: echo Beginning checkpoint to tape %C of %N locations
```

indicates what is done initially, before the device is used (B), when checkpointing is used to save (S) a database location to tape (T), for a database (D).

As another example, when executing a checkpoint on a database that spans multiple locations, one of the following commands is executed once for each location (WSTD for backup to tape, WSDD for backup to disk):

```
WSTD: ckcopyt %N %D BACKUP
```

```
WSDD: ckcopyd %D %A BACKUP
```

The commands instruct the checkpoint operation to call either the ckcopyt.bat or ckcopyd.bat batch command file to do the actual backup.

The checkpoint utility automatically substitutes the appropriate values for "%N," "%D," and "%A."

The ckcopyt.bat batch file calls the Windows backup backup command and passes it the name of the directory for the location, and other operating system flags.

Format of the Checkpoint Template File in UNIX

The checkpoint template file, cktmpl.def, uses the UNIX tar command. This file can be found in $II_SYSTEM/ingres/files.

Each line is a command preceded by a four-character code that instructs the checkpoint operation when to use the command.

By altering this file you can change how checkpoints are performed. You can add or delete flags from the tar commands or you can supply your own shell scripts to perform the backup and restore steps.

For example, the command:

```
BSTD: echo beginning checkpoint to tape %C of
      %N locations
```

indicates what is done initially, before the device is used (B), when the checkpoint operation is used to save (S) a database location to tape (T) for a database (D).

As another example, when executing a checkpoint on a database that spans multiple locations, the following command is executed once for each location:

```
PSTD: echo mount tape %N and press return;
      read foo;

WSTD: cd %D; /bin/tar cbf 20 %C *
```

The command instructs the checkpoint operation to save each location on a tape and to use the tar command with the parameter cbf 20. The checkpoint utility automatically substitutes the appropriate value for "%N," "%D," and "%C."

Alternate Checkpoint Template Files (UNIX and Linux)

UNIX: An alternate checkpoint template file, cktmpl_cpio.def, uses the UNIX cpio command to back up and restore the database files.

Linux: An alternate checkpoint template file, cktmpl_ocfs.def, is for use with the Oracle Cluster File System (OCFS) on Linux.

The alternate file can be found in $II_SYSTEM/ingres/files.

To use an alternate template file and override the default cktmpl.def template file, use the ingsetenv command. For example:

```
ingsetenv II_CKTMPL_FILE $II_SYSTEM/ingres/files/cktmpl_cpio.def
```

Format of the Checkpoint Template File in VMS

The checkpoint template file, cktmpl.def, can be found in $II_SYSTEM:[INGRES.FILES].

The checkpoint template file uses the four-letter key described above to begin each line. A line can specify an individual tape and disk handling command or the name of a user-written command file to provide more complex processing such as backing up all of a database's locations concurrently.

Defaults are provided so that sites using standard processing do not need to alter the checkpoint template file.

Here are some example lines from a cktmpl.def file:

```
WSTD: @ckp_to_tape "%N" "%D" "%C" "%F"
WSDD: @ckp_to_disk "%D" "%A"
WRTD: @rollfwd_from_tape "%N" "%D" "%C" "%F"
WRDD: @rollfwd_from_disk "%A" "%D"
```

Each of these example lines specifies the name of a command file and establishes requests for run-time information.

Backup and Recovery of the Master Database (iidbdb)

The iidbdb database is your Ingres installation's master database. It contains information about your installation as a whole, such as:

- Which databases exist in this installation
- Where user databases are located
- Which locations can be used for files
- Which users can access databases

The iidbdb also contains information about groups, roles, and database privileges defined for your site.

The iidbdb is journaled by default.

The iidbdb and Checkpointing

You should regularly checkpoint and journal the iidbdb database. Ckpdb and rollforwarddb are the supported utilities for recovering the iidbdb if it is lost or damaged for any reason. The system catalogs containing the installation information for groups, roles, and database privileges are stored in the iidbdb database and can only be recovered from backups.

Set Log_Trace Statement—Trace Log Writes

You can use the LOG_TRACE option of the SET statement to start and stop tracing of log file writes. Using this option requires the trace privilege.

Important! Do not use set log_trace alone as a debugging or tracing tool. Do not base applications on set log_trace output because it is not guaranteed to remain the same across releases. The support of set log_trace is not guaranteed in this or future releases.

To start tracing log writes, issue the following statement:

```
set log_trace;
```

To stop tracing log writes, issue the following statement:

```
set nolog_trace;
```

When you use SET LOG_TRACE during a session, you receive a list of the log records written during execution of your query, along with other information about the log. SET LOG_TRACE output includes:

- The length of the log and the amount of space reserved for its CLR. For more information on CLRs, see Log Space Reservation (see page 349).
- If the log write is a normal log record (do/redo) or a CLR.
- If the log record can be copied to the journal file.

  If the log is associated with a special recovery action.

Chapter 14: Calculating Disk Space

This section contains the following topics:

It is important to ensure that the Ingres installation has adequate disk space for storing the system executables and data tables. Disk space is also used during the execution of many commands.

This chapter discusses how to calculate the disk space needed for the various files and operations of an Ingres installation.

Space Requirements for Tables

This section defines terms applicable to page size and gives calculations for estimating the amount of disk space needed for tables. These are approximations—your table can be much larger, depending on compression and the size of key values.

The calculations are based on newly modified tables. Using the number of rows in the table to determine table size becomes less accurate after data has been deleted or added.

VDBA provides a calculation tool that allows you to calculate disk space requirements for any storage structure quickly and easily. For procedures, see online help.

Calculate Space Requirements for Heap Tables

Note: If rows in the table span pages, use the procedure in Calculate Space Requirements When Rows Span Pages (see page 403) instead.

Use the following procedure to determine the amount of space needed to store the data in a heap table:

1. Create the table.
2. Determine the number of rows that fit on a page.

   select tups_per_page from iitables where table_name = 'tablename';
3. Determine the total number of pages needed if the table is a heap.

   total_heap_pages = num_rows / tups_per_page

Calculate Space Requirements for Hash Tables

Note: If rows in the table span pages, use the procedure in Calculate Space Requirements When Rows Span Pages (see page 403) instead.

Follow these steps to determine the amount of space needed to store the data in a hash table.

1. Create the table and modify it to hash.
2. Determine the number of rows that fit on a page, adjusted for the data page fillfactor to be used.

   select tups_per_page * *table_dfillpct/100 from iitables where table_name* = 'tablename';
3. Determine the total number of pages needed for a hash table.

   total_hash_pages = (*num_rows*/(*tups_per_page*)

   Note: Because hashing does not guarantee an equal distribution of rows, the actual number of pages required can be greater than calculated above.

Calculate Space Requirements for ISAM Tables

Follow these steps to determine the amount of space needed to store the data in an ISAM table:

1. Create the table and modify it to ISAM.
2. Determine the number of rows that fit on a page (adjusted for data page fillfactor) and the number of keys that fit on an index page.

   *select tups_per_page * table_dfillpct/100, keys_per_page from iitables where* table_name = *'tablename';*
3. Determine the number of data pages needed for the table:

   data_pages = (num_rows / tups_per_page)

   Note: When rows span pages, determine the number of data pages using the calculation in Calculate Space Requirements When Rows Span Pages (see page 403) instead.
4. Determine the number of index pages needed for the table:

   index_pages = data_pages / keys_per_page

   Note: When rows span pages, use the following calculation instead:

   index-pages = num_rows / keys_per_page
5. Determine the total number of pages needed for the table. The total includes data pages and index pages. The total number of allocated pages in an ISAM table is never less than keys_per_page.

   total_isam_pages = data_pages + index_pages

   if (total_isam_pages < keys_per_page)

   total_isam_pages = keys_per_page

Calculate Space Requirements for B-tree Tables

Follow these steps to determine the amount of space needed to store the data in a B-tree table:

1. Create the table and modify it to B-tree.
2. Determine the number of rows that fit on a page, the number of keys that fit on an index page, and the number of keys that fit on a leaf page (adjusted by the appropriate fillfactors):

   *select tups_per_page * table_dfillpct/100, keys_per_page * table_ifillpct/100, keys_per_leaf * table_lfillpct/100 from iitables where table_name = 'tablename';*

3. Determine the number of leaf pages needed. Save the remainder of the division because it is used later:

   leaf_pages = (*num_rows*/*keys_per_leaf*)

   remainder = modulo (*num_rows* / (*keys_per_leaf*)

4. Determine the number of data pages needed.

   data_pages = *leaf_pages* * (*keys_per_leaf* / *tups_per_page*)

   Note: When rows span pages, determine the number of data pages using the calculation in Calculate Space Requirements When Rows Span Pages (see page 403) instead.

5. If the remainder from Step 3 is greater than 0, adjust the number of leaf and data pages:

   a. *leaf_pages* = *leaf_pages* + 1

   b. Round the division **up** to the nearest integer:

      data_pages = *data_pages* + (*remainder* / *tups_per_page*)

   Note: When rows span pages, Step 5b does not apply.

6. Determine the number of sprig pages.

   sprig_pages: The number of index pages that have leaf pages as their next lower level:

   a. If *leaf_pages* <= *keys_per_page*, then *sprig_pages* = 0

   b. Otherwise, calculate as follows, and round **up** to the nearest integer:

      sprig_pages = (*leaf_pages* / *keys_per_page*)

7. Determine the number of index pages.

   index_pages: The number of index pages that are not sprig pages. This is done iteratively. Do the following if *sprig_pages* > *keys_per_page*:

   ```
   x = sprig_pages
   do
     {
     x = x / keys_per_page
     index_pages = index_pages + x
     }
   while (x > keys_per_page>
   ```

8. Determine the total space required. The total includes data pages, leaf pages, sprig pages, and index pages.

   total_btree_pages = *data_pages* + *leaf_pages* + *sprig_pages* + *index_pages*

Calculate Space Requirements When Rows Span Pages

Follow these steps to determine the amount of space needed to store the data in a table with rows that span pages:

1. Determine the number of pages per row, as follows:

   pages_per_row = *row_size* / *max row size*

   where *max_row_size* is the maximum row size for the table, as shown in Maximum Row Size Per Page Size (see page 403).

   Round up to the nearest integer.

2. Determine the number of data pages needed for the table, as follows:

   data_pages = *num_rows* * *pages_per_row*

Maximum Row Size Per Page Size

Table rows span pages if the row size is greater than the maximum row size for the table page size, as shown in this table:

Page Size	Max Row Size
2048 (2 KB)	2008 bytes
4096 (4 KB)	3988 bytes
8192 (8 KB)	8084 bytes
16384 (16 KB)	16276 bytes
32768 (32 KB)	32660 bytes

Page Size	Max Row Size
65536 (64 KB)	65428 bytes

Space Requirements for Compressed Tables

Table size for compressed tables is not possible to determine by an algorithm because the number of fields that can be compressed, and what percentage they can be compressed, differ for every table.

To get any sort of estimate, you must guess the amount by which each record, on the average, can be compressed. Use this estimated record width to determine the size of the table as if it were uncompressed, using the rules set forth above.

Tracking of Used and Free Pages

The DBMS space management handles used and free page tracking. A table uses a combination of a single free header page (H) and one or more free map pages (M) to track free and used pages.

Each free map page can track 16,000 pages, recording whether the page is free or used. As tables are allowed to grow past 16,000 pages, there can be more than one free map page in a table.

Free map pages are tracked by the free header page, whose location is recorded in the system catalog entries for the table.

Free header and free map pages are additional pages required for each table page count.

Note: All tables in the database can grow by a minimum of two pages. One free map page is added per 16,000 pages.

In VDBA, to view the graphical display of the pages, select a table and select the Pages tab.

Note: For B-tree tables, all empty disassociated data pages and any pages on the old free list are marked as “used.” The only way to reclaim this space is to select Shrink B-tree Index in the Modify Table Structure dialog in VDBA, or use the MODIFY TABLE TO BTREE statement in SQL. For more information, see the *SQL Reference Guide*.

Calculation of Allocated Table Size

VDBA automatically calculates table size based on the number of allocated pages. Using VDBA, select a table and select the Pages tab to view the pages property sheet.

Alternatively, the allocated_pages field in the iitables standard catalog can be used to calculate table size based on the number of allocated pages. You can calculate:

- The size of the table on disk as:

  iitables.allocated_pages * PageSize

- The number of free pages left in the table as:

  iitables.allocated_pages – iitables.number_pages

Space Requirements for Journal Files

Journal files are created in the database's journal directory. This is a single directory.

The archiver moves log file records for journaled tables affected by committed transactions to the database's journal file during periodic sweeps through the log file. The journal files are never directly connected to user sessions—they are written only by the archiver.

The following conditions cause a new journal file to be started:

- Once the journal file reaches a certain size (currently about 8 MB) a new journal file is started.
- Each checkpoint starts a new journal file upon its successful completion, because the journal files contain records of changes made after a specific checkpoint was taken.

You can delete old, unneeded journal files using VDBA. For more information, see Setting Checkpoints in online help.

To accomplish this task at the command line, use the ckpdb system command. For more information, see the *Command Reference Guide*.

Space Requirements for Modify Operations

Modify operations require additional working disk space because a new version of the table must be built before the old table can be removed. In most cases, modify operations require about two or three times more space than the original table size. This is only an approximation; the amount of disk space actually needed can vary.

The free disk space is also required during modify to relocate and modify to reorganize operations.

For information about relocating and changing the location of storage structures, see the Modifying Storage Structures topic in online help for VDBA. For details on the DBA use of these operations, see Techniques for Moving a Table to a New Location (see page 58).

In SQL, you can accomplish these tasks with the MODIFY TO REORGANIZE and MODIFY TO RELOCATE statements. For more information, see the *SQL Reference Guide*.

The maintain_locations privilege is needed to perform the operation on location objects in VDBA or to issue the MODIFY LOCATION statement in SQL. The maintain_locations privilege allows users to do the following:

- Control the allocation of disk space
- Create new locations or allow new locations to be created
- Modify or remove existing locations

Factors Affecting Space Requirements for Modify Operations

The following are important factors that affect disk space requirements:

- You need at least twice the disk space of the table ("2X"), one copy of the original table and one copy of the new table.
- The new table size can be increased if an index is being added. Conversely, space can be freed if an index is no longer necessary. Index space can vary widely depending on the size of the key.
- If you are modifying to a sorted structure (ISAM, B-tree) or to hash, an additional copy of the original table is needed, thus requiring three times the disk space of the table ("3X").
- If you are modifying a compressed table, calculate disk space based on the uncompressed size. In the worst case, this is the row size times the number of rows.

  Usually going from a compressed structure to an uncompressed structure increases the table size, and going the other way decreases its size. The amount of change cannot be predicted and is dependent on the data in the table. If many NULL values are present and if many string fields have trailing blanks, the use or omission of compression is very noticeable.
- Fill factor, minpages, leaffill, and nonleaffill also play a role in the resulting table size. For details, see Options to the Modify Procedure (see page 210).

Summary of Space Requirements for Modify Operations

The following table provides a summary for estimating disk space requirements.

In the table, "O+N" (Original+New tables) corresponds roughly to twice the table size (2X) and "O+N+S" to three times the table size (3X). The space required can be affected by whether an index is added ("I" in the table) or existing index space freed ("U" in the table).

Original Table Structure	**Modified to**			
	Heap	**Hash**	**ISAM**	**B-tree**
Heap	O+N	O+N+S	O+N+S+I	O+N+S+I
Hash	O+N	O+N+S	O+N+S+I	O+N+S+I
ISAM	O+N-U	O+N+S-U	O+N+S	O+N+S
B-tree	O+N-U	O+N+S-U	O+N+S	O+N+S

Legend:

O = Original table size
N = New table size
S = A sort is required
I = Index is being added
U = Space freed because an index is no longer necessary

Note: Remember that numerous factors contribute to the actual disk space used in a particular modify operation. Additional factors include compression and the various fill values.

Space Requirements for Sorts

Sorting occurs commonly during many index, copy, and modify operations. (Sorting also occurs in the processing of the equivalent SQL statements.)

When the size of the sort requires disk space, temporary work locations are used.

A default work location area is defined during installation.

The disk space required for sorting depends on how much sorting needs to be done. If the table to be sorted is badly out of sorted order, more space can be used than if it is in nearly sorted order.

For a nearly sorted table, the amount of work location space is equal to the uncompressed size of the table.

For a table that is badly out of order, the maximum work location space is two times the uncompressed size of the table, and the space required per location can be estimated by the formula:

```
(2 * uncompressed_table_size) / number_of_work_locations
```

Insufficient Sort Space

If any work location runs out of disk space during a sort, the sort fails and the associated transaction is aborted.

To correct this situation, you can add additional work locations or provide more space on the device that filled (by removing or relocating unneeded files). Alternatively, the device that filled can be dropped through the SET WORK LOCATIONS statement.

For information about work locations and the SET WORK LOCATIONS statement, see Work Locations (see page 35). For a discussion on deleting unneeded files, see the chapter "Maintaining Databases."

Orphaned Sort Files

Sort files can be left in work locations after certain types of failures.

In VDBA, use the "verify database" procedure to remove these orphaned files. For more information, see Verifying a Database in VDBA online help.

To accomplish this task at the command line, use the verifydb command. For more information, see the *Command Reference Guide*.

Factors Affecting Sort Performance

The use of multiple work locations does not generally affect overall sort performance.

In UNIX, the performance of very large sorts can be affected by the amount of available operating system cache memory. While most aspects of server performance are largely unaffected by the OS cache size, sorts employ the OS cache, as well as the Ingres DBMS Server DMF cache. Sorting time can sometimes be improved by configuring additional OS memory.

Chapter 15: Improving Database and Query Performance

This section contains the following topics:

This chapter contains information on how to improve and optimize query and database performance. Good performance requires planning and regular maintenance.

The techniques and procedures in this chapter may help you to solve a performance problem yourself or to accurately define the problem if you must call customer support.

Note: This chapter assumes that Ingres is running satisfactorily. If you are encountering problems with the operation of Ingres, first see the *System Administrator Guide* for troubleshooting information.

Locking and Concurrency Issues

If your performance problem occurs in a multi-user environment or if the query runs slowly or hangs intermittently, you can have a concurrency problem.

Concurrency problems occur when several users access the same tables and at least one is a writer. If your query needs to access objects that are locked, the session waits indefinitely for locks to be released unless the lockmode timeout is set or a deadlock occurs.

Lock Waits and Performance

To monitor locks, use the Lock Information branch of the Performance Monitor window in VDBA to monitor lock waits. For details, see Viewing Performance Information in online help. The Performance Monitor can also be accessed by choosing Ingres Visual Performance Monitor from the Ingres menu.

If you find lock waits, identify the queries that are holding locks on the resources you are waiting to access. You must modify your locking strategy to avoid future problems.

Pay particular attention to:

- maxlocks
- readlock = nolock
- timeout
- set lock_trace command

If the lock being waited on was created as the result of lock escalation, your system is configured with too few system-wide locks. This is a configuration issue; see the *System Administrator Guide*.

If lock escalation occurs because too many locks are taken on a given table's pages, a SET LOCKMODE statement can be issued to increase this threshold. The default is 10 before escalation occurs. For more information, see the chapter "Understanding the Locking System."

Multi-query Transactions and Performance

Remember that a transaction accumulates locks on resources until you roll back or commit. A transaction that is waiting for locks, or that is not waiting for a lock but nevertheless seems unusually slow, can be using excessive server or system resources.

Here are suggestions:

- Keep your transactions as short as possible.
- Commit your transactions quickly:
  - You create large multi-query transactions (MQTs) unless you use SET AUTOCOMMIT ON or COMMIT after each statement. Statements accumulate as one multi-query transaction until you commit.
  - MQTs must not include prompts that hang the transaction until a user responds, or sleeps that prevent your transaction from being released quickly.
- Avoid bottlenecks in your transaction such as:
  - Insert to heap table with secondary indexes
  - Counter table updates
  - Iterative deletes
  - Unbounded long iterations

Overflow and Performance

Overflow chains slow concurrent performance. Overflow pages are attached to the main data page if a record must be added to a full main page. The query that touches one main data page must now touch that page plus each associated overflow page. This increases I/O, cause concurrency problems, and uses up locking system resources.

Here are suggestions:

- Monitor overflow chains.

  Check the number of overflow pages for your tables and secondary indexes. To monitor overflow in VDBA, select a table or secondary index in the Database Object Manager window, and click the Pages tab. Use the legend to interpret the information displayed.

  If the number of overflow pages is greater than 10-15% of the number of data pages, expect performance degradation.

- Check for duplicate keys. Overflow problems are often caused by them.
- Consider trying a different storage structure. Some table structures create long overflow chains when much new data is added. For details, see Storage Structure and Overflow (see page 415).
- Decrease overflow

  Here are ways to decrease overflow and improve concurrency:

  – Use unique keys.

  – Modify the table to reorganize it; with a B-tree structure, simply specify the Shrink B-tree Index option.

  – Consider tailoring the table’s fill factor.

For additional information, see the sections on overflow and fill factor in the chapters “Choosing Storage Structures and Secondary Indexes” and “Maintaining Storage Structures.”

Storage Structure and Overflow

Here are overflow considerations for each storage structure:

- Heap—Heap tables are created as one main page with an overflow chain. There is no overflow management.
- Hash—Overflow pages occur in a newly modified table if the key is repetitive; this is normal but undesirable. Check a freshly modified table. If there is overflow, consider using ISAM instead.
- ISAM—ISAM has a fixed index that can cause long overflow chains. Modify frequently or use B-tree for a non-static table. Use heap structure for large bulk updates and modify back to ISAM to avoid update performance problems.
- B-tree—No overflow if there are no duplicate keys, so consider making keys unique. Overflow occurs only at the leaf level and only when 2K pages are used. Use the Shrink B-tree Index option to reorganize it. Use heap structure for bulk loads, modify to B-tree.

Set Statements and Locking Strategy

There are a variety of SET statements you can use to manage your locking strategy.

Be sure you are using user-defined lockmodes and isolation levels to their fullest to avoid concurrency and deadlock. For assistance with strategy, see the chapter "Understanding the Locking System." For command syntax, see your query language reference guide.

Pay particular attention to:

- Deadlock
- Lock_trace flag
- Maxlocks
- Readlock = nolock
- Timeout

For additional information on the use of the SET statement to customize the query environment, see the *System Administrator Guide*.

Database Maintenance Issues

If your query used to run quickly and is now slower, or the speed of the query changes depending on the constants specified in the WHERE clause, your problem can be poor database maintenance.

To optimize performance, set up maintenance procedures that run DBA utilities.

The following features are especially useful in tracking performance problems:

- Optimization
- Modification of table and index structure
- System modification
- Verification

For discussions of maintenance issues, see the chapters “Maintaining Databases,” “Maintaining Storage Structures,” and “Using the Query Optimizer.”

Optimization and Performance

The optimization feature collects statistics that are used by the query optimizer to determine the best query execution plan (QEP) to use for your queries.

Follow these optimization guidelines:

- Periodically run optimization on all your databases to generate statistics for columns that are keys or indexed. List the other columns you need as an argument to this command.
- Run full optimization statistics on columns referenced in WHERE clauses of strategic queries that are having problems.
- For very large tables, create statistics based on sample data.
- When there are significant changes to your data distribution, run optimization on the affected columns.
- Do not collect excessive statistics, because you build up large optimizer tables with unused data.
- Run system modification after every optimization.

To perform optimization, use the optimizedb command or in VDBA use the Optimize Database dialog.

Table and Index Modification and Performance

You can modify a table or index to:

- Reorganize data on new data pages
- Free deleted record space
- Reduce overflow chains
- Adjust the fill factor

Use the Shrink B-tree Index option (or MODIFY TO MERGE statement) to:

- Reorganize index pages of B-tree tables
- Reduce overflow chains

Use the Change Location option (or MODIFY TO RELOCATE statement) to move your tables to balance disk access.

To perform modification, use the MODIFY statement or, in VDBA use the Modify Table Structure and Modify Index Structure dialogs.

System Modification and Performance

The system modification feature modifies system catalogs to predetermined storage structures.

- Run system modification on the iistatistics system catalog after optimization.
- Run system modification on ii_rcommands if you create and update a lot of Report-Writer reports.
- Regularly using system modification reduces overflow in your system catalogs. Run it often if catalog changes are frequent due to development or if you use many CREATE or DROP statements in your applications.

To perform system modification, use the sysmod command or, in VDBA, use the System Modification dialog

Verification and Performance

Use the verification utility to:

- Destroy or list unrequired disk files, expired tables, or temporary tables in your database
- Clean up fragmented disk space

To perform verification, use the verifydb command or, in VDBA, use the Verify Database dialog.

Design Issues and Performance

Good query performance requires planning.

Carefully plan the design of the following:

- Storage structures and indexes
- Keys
- Queries

For help in identifying performance issues, see the chapters “Ensuring Data Integrity,” “Maintaining Databases,” “Maintaining Storage Structures,” and “Using the Query Optimizer.”

Other important design issues are:

- Database design
- Validation checks and integrities
- Grants and views
- Application design

Hierarchy for Diagnosing Design-based Performance Problems

A thorough performance analysis must include each item in the following list. Areas are listed in the order of greatest gain. For example, if your database design is flawed, perfect server configuration cannot help you avoid query performance problems.

1. Database design
2. Storage structures and index design. See the chapter "Choosing Storage Structures and Secondary Indexes."
3. Key design. See the chapter "Choosing Storage Structures and Secondary Indexes."
4. Constraints. See the chapter "Managing Tables and Views."
5. Validation checks and integrities. See the *Security Guide* and the chapter "Ensuring Data Integrity" in this guide.
6. Grants and views. See the *Security Guide*.
7. Query design.
8. Application design.
9. Concurrency. See the chapter "Understanding the Locking System" and the *System Administrator Guide*.
10. DBA utilities and maintenance. See the chapter "Maintaining Databases."
11. Operating system resources and tuning. See the *System Administrator Guide*.
12. Server configuration. See the *System Administrator Guide*.

Storage Structures and Index Design and Performance

Choosing the correct table storage structure for your needs can improve concurrency and query performance. Remember that there is no substitute for testing and benchmarking your queries.

For tips on choosing storage structures and advantages and disadvantages of the various storage structures, see the chapter "Choosing Storage Structures and Secondary Indexes." For information on modifying and compressing storage structures and a discussion of overflow, see the chapter "Maintaining Storage Structures."

Key Design and Performance

Key design is a complex subject. For additional information on keys, see the chapter "Choosing Storage Structures and Secondary Indexes."

Characteristics of Good Keys

Good keys have the following features:

- Use columns referenced in the WHERE clauses and joins of your queries
- Are unique

  Always document reasons for maintaining non-unique keys.

All keyed storage structures can enforce unique keys. They are:

- Short
- Static
- Non-nullable

Characteristics of Bad Keys

Bad keys have the following features:

- Wide
  - Use wide keys with caution.
  - You get fewer rows per page.
  - Evaluating the hash function takes more time with wide keys.
  - A wide key deepens the index level of B-tree and ISAM logarithmically, with respect to key width. B-tree is the least affected table structure.
  - Consider using a surrogate key as an alternative.
- Non-static

  Updating the index can slow performance.
- Non-uniform duplication

  A mix of high and low duplication can cause inconsistent query performance.
- Sequential
  - Sequential keys must be used with care.
  - ISAM tables can be lopsided and the overflow chains can cause concurrency problems.
  - Control sequential key problems with a frequent modify schedule.

Multi-Column Keys and Performance

Multi-column keys have special issues. If used improperly in your query, the key cannot be used and the search does a full-table scan.

Keep the following in mind:

- Use the most unique and frequently used columns for the left member of a multi-column key.
- Searches on B-tree and ISAM tables must use at least the leftmost part of a multi-column key in a query, or a full-table scan can result.
- Searches on hash tables must use an exact match for the entire key in the query, or a full-table scan can result.
- Optimizer statistics are approximated by adding the statistics of the columns making up a multi-column key.

Surrogate Keys and Performance

When you use a short surrogate or internal key to replace a bad key, or because there is no good key, consider the performance trade-offs. The set processing of data includes the overhead of deriving the key.

Surrogate key types include:

- Natural

  Universal (a social security number or zip code are examples)
- Environmental

  These are local to an organization, like an employee number.
- Design artificial. These are:
  - Local to an application
  - Hard to remember
  - Hard for users to understand
  - Can be hidden from users

Query Design and Performance

Query design is a complex subject. Following these tips will improve the performance of your queries:

- Conversion joins are joins where two columns of different data types are joined in a query, either explicitly or implicitly. These joins are frequently the result of database design problems and must be avoided.
- Avoid using function joins.
  - Functions in the WHERE clause force a full-table scan.
  - Control uppercase and lowercase, and so on, at input time.
- Some complex OR queries can be rewritten as unions.
- Evaluate QEPs for critical queries:

  Can large table scans be avoided?
  - Is an additional index needed?
  - Are Cartesian products with large tables used?
  - Are function joins used?
- Use repeated queries for queries that are used many times.
- Do not forget to commit. Consider using SET AUTOCOMMIT ON.

Information Needed By Customer Support

If you have worked through the query performance evaluation and your problem is not resolved, call customer support. Before calling, follow these two procedures:

- Isolate and analyze the suspect query
- Create a test case

Isolate and Analyze the Problem Query

To determine whether the problem is due to the user interface, the query itself, or a software bug, follow these steps:

1. Isolate a poorly performing query from your user interface using the trace flag set printqry, which prints queries before they are optimized and executed. Identify the query that seems to hang.

   For details on setting printqry, see the *System Administrator Guide*.

   Execute the query in a terminal monitor or from within the VDBA SQL Scratchpad window, and determine if performance is the same. If performance is only a problem when the query is executed from the user interface, you have identified an application problem. If performance is the same, continue.

2. In a terminal monitor, issue the following statements to display the QEP without running the query:

   ```
   set qep;
   set optimizeonly;
   ```

   Now, execute your query and save the output to a file for examination. After running the query, exit the terminal monitor session or turn query execution back on using:

   ```
   set nooptimizeonly;
   ```

   For details on these set statements, see the *System Administrator Guide*.

3. Review the Design Issues section and evaluate the QEP for your query. For example, you can look for:

   - Large table scans that can be avoided
   - An additional index that is needed
   - Cartesian products with large tables
   - Function joins
   - If you are not able to identify your problem and suspect a software bug, submit your query and a test case to customer support.

Create a Test Case

To create a test case, follow these steps:

1. Verify that you are using the most recent release of Ingres available for your platform.
2. Collect the information customer support needs to duplicate your problem.

   Customer support needs the following information in ASCII files that you can send by e-mail, UUCP, or on a tape:

   - The exact query that causes the error to occur
   - The QEP generated by the problem query
   - Dump optimizer statistics for all the tables in the query (use the Direct Output to Server File option in the Display Statistics dialog in VDBA, or the statdump command with the -o flag)
   - The help table tablename information for all the tables that the query references (or equivalent information obtained from within VDBA)
   - The help index indexname information for all secondary indexes of tables in the query (or equivalent information obtained from within VDBA)
   - The help permit on table *tablename* information for grants on all the tables in the query (or equivalent information obtained from within VDBA)
   - A query of the system catalogs for information about each table. Look at iirelation and select relpages, reltups, relmain, and relprim, where the relid is equal to each table and index in the query.

The create scripts and data for all the tables, indexes, and grants that the query references. When generating the scripts, you must specify the Create Printable Data Files option. For information on generating these scripts, see the chapter “Loading and Unloading Databases.”

Appendix A: System Catalogs

This section contains the following topics:

This appendix describes the Standard Catalog Interface catalogs, the Extended System catalogs, and the DBMS System Catalogs.

This appendix describes the catalogs for Ingres 9.2. For catalog formats of other Ingres releases, see the appropriate documentation set.

Note: The DBMS System Catalogs and Extended System Catalogs are unsupported and can change at any time. Information about these catalogs is provided solely for your convenience. In providing this information, the company makes no commitment to maintain compatibility with any feature, tool, or interface. The company does not provide support, either through customer support or release maintenance channels, for the resolution of any problems or bugs arising from the use of unsupported features, tools, or interfaces.

Standard Catalog Interface

Each database has a set of system catalog tables that store information (for example, metadata) required by Ingres. The System Catalog Interface (SCI) is a set of views on top of these system catalog tables, and a set of tables that can be queried through SQL statements and therefore used in applications to access (but not update) information about the database.

If you are developing applications that need to query the system catalogs, you must use the Standard Catalog Interface, so that your applications will be upwardly compatible.

All database users can read the Standard Catalog Interface catalogs, but only a privileged user can update them.

To display the underlying view, use the SQL statement HELP VIEW. To display the format of catalogs, use the SQL statement HELP (see page 427).

The length of character fields, as listed in the length column, is a maximum length; the actual length of the field may be installation dependent. The values are left-justified and the columns are non-nullable.

Unless otherwise stated, dates are displayed as Ingres dates, that is, 25-byte character strings with the following format:

```
yyyy_mm_dd hh:mm:ss GMT
```

When developing applications that access SCI, storage should be allocated based on the length shown in the Data Type column in the descriptions below.

Example of HELP VIEW and HELP Statements

The following HELP VIEW statement displays information about the iisynonyms view:

```
help view iisynonyms
```

Example output:

```
View:             iisynonyms
Owner:            $ingres
Check option:     off

View Definition:
create view  iisynonyms(
             synonym_name,
             synonym_owner,
             table_name,
             table_owner)
as
select synonym_name,
       synonym_owner,
       relid,
       relowner
from   "$ingres".iirelation,
       "$ingres".iisynonym
where  reltid   = syntabbase
and     reltidx = syntabidx
```

The following HELP statement displays the format of the iisynonyms catalog:

```
help iisynonyms
```

Example output:

```
Name:                 iisynonyms
Owner:                $ingres
Created:              17-dec-2007 10:11:56
Type:                 system catalog
Version:              II9.0

Column Information:
                                                           Key
Column Name                Type       Length Nulls Defaults Seq
synonym_name               char         32    no     no
synonym_owner              char         32    no     no
table_name                 char         32    no     yes
table_owner                char         32    no     yes
```

Standard Catalogs for All Databases

The standard catalogs for all databases are as follows:

iiaccess	iialt_columns	iiaudittables
iicolumns	iiconstraint_indexes	iiconstraints
iidb_comments	iidb_subcomments	iidbcapabilities
iidbconstants	iidistcols	iidistschemes
iievents	iifile_info	iihistograms
iiindex_columns	iiindexes	iiingres_tables
iiintegrities	iikeys	iikey_columns
iilog_help	iilpartitions	iimulti_locations
iipermits	iiphysical_tables	iiprocedures
iiproc_access	iiproc_params	iirange
iiref_constraints	iiproc_rescols	iirules
iisecurity_alarms	iiregistrations	iisequences
iistats	iisession_privileges	iitables
iiviews	iisynonyms	

iiaccess Catalog

The iiaccess catalog holds information about permissions on tables, views, and indexes.

Column Name	Data Type	Description
table_name	char(32)	Name of the table, view, or index
table_owner	char(32)	Owner of the table, view, or index
table_type	char(1)	T—Base table V—View I—Index
system_use	char(1)	S—System catalog object U—User object G—Generated
permit_user	char(32)	Name of grantee or empty string
permit_type	char(64)	Privilege granted

iialt_columns Catalog

All columns defined as part of an alternate key have an entry in iialt_columns.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The name of the table owner
key_id	integer	The number of the alternate key for this table
column_name	char(32)	The name of the column
key_sequence	smallint	Sequence of column in the key, numbered from 1

iiaudittables Catalog

The iiaudittables catalog provides a list of currently registered security audit log files for the database.

Column Name	Data Type	Description
table_name	char(32)	The name of the virtual security audit table
table_owner	char(32)	The name of the table owner as determined by the register table statement
audit_log	char(256)	The full file name specification of the underlying security audit log
register_date	char(25)	The date and time the audit table was registered

iicolumns Catalog

For each queriable object in the iitables catalog, there are one or more entries in the iicolumns catalog. Each row in iicolumns contains the information on a column of the object. Iicolumns is used by Ingres tools and user programs to perform dictionary operations and dynamic queries.

Column Name	Data Type	Description
table_name	char(32)	The name of the table.
table_owner	char(32)	The owner of the table.

Column Name	Data Type	Description
column_collid	smallint	The column's collation ID. Valid values are: -1 The default 1 for unicode 2 for unicode_case_insensitive 3 for sql_character
column_name	char(32)	The name of the column
column_datatype	char(32)	The data type of the column: INT INTEGER SMALLINT FLOAT REAL DECIMAL DOUBLE PRECISION C CHAR CHARACTER VARCHAR LONG VARCHAR BYTE LONG BYTE TEXT MONEY INGRESDATE ANSIDATE TIME TIMESTAMP INTERVAL
column_length	integer	The length of the column. Displays the precision for decimal data, zero for money and date
column_scale	integer	Displays the scale for decimal data type, zero for all other data types
column_nulls	char(1)	Y if the column can contain null values, N if the column cannot contain null values
column_defaults	char(1)	Y if the column has a default value when a row is inserted, N if not
column_sequence	integer	The number, from 1, of the column in the corresponding table's create statement

Column Name	Data Type	Description	
key_sequence	integer	The order, numbered from 1, of this column in the primary key for a table. 0 if this column is not part of the primary key	
sort_direction	char(1)	A for ascending; used when key_sequence is greater than 0	
column_ingdatatype	integer	Contains the internal numeric representation of the column's external data type. If the value is positive, the column is not nullable. If the value is negative, the column is nullable. If the installation has user-defined data types (UDTs), this column contains the data type that the UDT is converted to when returned. The data types and their corresponding values are:	
		INTEGER	30/-30
		FLOAT	31/-31
		C	32/-32
		TEXT	37/-37
		INGRESDATE*	3/-3
		DECIMAL	10/-10
		MONEY	5/-5
		CHAR	20/-20
		VARCHAR	21/-21
		LONG VARCHAR	22/-22
		BYTE	23/-23
		LONG BYTE	25/-25
		TABLE_KEY	12/-12
		OBJECT_KEY	11/-11
		ANSIDATE	4/-4

Column Name	Data Type	Description	
		TIME WITHOUT TIMEZONE	6/-6
		TIME WITH TIMEZONE	7/-7
		TIME	8/-8
		TIMESTAMP WITHOUT TIMEZONE	9/-9
		TIMESTAMP WITH TIMEZONE	18/-18
		TIMESTAMP	19/-19
		INTERVAL YEAR TO MONTH	33/-33
		INTERVAL DAY TO SECOND	34/-34
		*Returned to applications as a string.	
column_internal_ datatype	char(32)	The internal data type of the datatype column: CHAR C VARCHAR TEXT INTEGER FLOAT DATE DECIMAL MONEY TABLE_KEY OBJECT_KEY If the installation has user-defined data types, this column contains the user-specified name.	
column_internal_ length	integer	The internal length of the column. 0 if the data type is date or money Does not include the null indicator byte for nullable columns or the 2-byte length specifier for varchar and text columns	
column_internal_ ingtype	smallint	The numeric representation of the internal data type.	

Column Name	Data Type	Description
		See column_ingdatatype for a list of valid values. If the installation has user-defined data types, this column contains the user-specified data type number.
column_system_ maintained	char(1)	Y if system-maintained N if not system-maintained
column_updateable	char(1)	Y if the column can be updated N if the column cannot be updated Blank if unknown
column_has_default	char(1)	Y if the column is defined with a default value, N if the column is defined as not default, U if the column is defined without a default Blank if unknown
column_default_val	varchar(1501)	The value of the default if the column has one Null if the default is not specified, NOT DEFAULT, or not known It contains surrounding and embedded quotes for character defaults, per ISO Entry SQL92 semantics.
security_audit_key	char(1)	Y if column is a security audit key N if column is not a security audit key

iiconstraint_indexes Catalog

The iiconstraint_indexes catalog contains information about constraint indexes.

Column Name	Data Type	Description
constraint_name	char(32)	The name of the constraint
schema_name	char(32)	The name of the schema
index_name	char(32)	The name of the index

iiconstraints Catalog

The iiconstraints catalog contains constraint information.

Column Name	Data Type	Description
constraint_name	char(32)	The name of the constraint
schema_name	char(32)	The name of the schema
table_name	char(32)	The name of the table
constraint_type	char(1)	The type of constraint: U if Unique P if Primary C if Check R if References Blank
create_date	char(25)	The date the constraint was created
text_sequence	integer8	The sequence number, from 1, for the text_segment
text_segment	varchar(240)	The text of the constraint definition
system_use	char(1)	U if the object is a user object G if generated by the system for the user. A status of G is used for constraints or views with check option.

iidb_comments Catalog

The iidb_comments catalog contains table comments.

Column Name	Data Type	Description
object_name	char(32)	The name of the table, view or index
object_owner	char(32)	The owner of the table, view or index
object_type	char(1)	Always T
short_remark	char(60)	The text of the short remark Blank if none
text_sequence	integer8	Always 1; the sequence number of the long_remark
long_remark	varchar (1600)	The text of the long remark If none, a zero-length string

iidb_subcomments Catalog

The iidb_subcomments catalog contains column comments.

Column Name	Data Type	Description
object_name	char(32)	The name of the table, view or index
object_owner	char(32)	The owner of the table, view or index
subobject_name	char(32)	The name of the column
subobject_type	char(1)	Always C
short_remark	char(60)	The text of the short remark Blank if none
text_sequence	integer8	Always 1; the sequence number of the long_remark.
long_remark	varchar(1600)	The text of the long remark If none, a zero-length string

iidbcapabilities Catalog

The iidbcapabilities catalog contains information about the capabilities provided by the DBMS.

Column Name	Data Type	Description
cap_capability	char(32)	Contains one of the values listed in the capability column of the table below.
cap_value	char(32)	The contents of this field depend on the capability; see the Value column in the table below.

The cap_capability column contains one or more of the following values:

Capability	Value
COMMON/SQL_LEVEL	Deprecated. Use OPEN/SQL_LEVEL
DB_DELIMITED_CASE	The case of delimited identifiers: LOWER for lowercase (Ingres setting) MIXED for mixed case (ISO Entry SQL92 setting) If MIXED, an identifier must be enclosed

Capability	Value
	in double quotes to maintain its original case; otherwise, it is converted to uppercase.
DB_NAME_CASE	The case of regular identifiers: LOWER for lowercase (Ingres setting) UPPER for uppercase (ISO Entry SQL92 setting)
DB_REAL_USER_CASE	The case of user names as retrieved by the operating system. LOWER for lowercase (Ingres setting) MIXED for mixed case UPPER for uppercase
DBMS_TYPE	The type of DBMS the application is communicating with. Valid values are the same as those accepted by the WITH DBMS = clause. Examples: INGRES (default) STAR RMS.
DISTRIBUTED	Y if the database is distributed N if database is local
ESCAPE	Y if DBMS supports the ESCAPE clause of the LIKE predicate in the WHERE clause N if ESCAPE is not supported
INGRES	Y if the DBMS supports all aspects of Release 6 and Ingres (the default) N if not
INGRES/SQL_LEVEL	Version of SQL supported by the DBMS Examples: 00600 6.0 00601 6.1 00602 6.2 00603 6.3 00604 6.4 00605 OpenIngres1.x 00800 OpenIngres 2.0 and Ingres II 2.0 00850 Ingres II 2.5 00860 Ingres 2.6

Capability	Value
	00902 Ingres r3 00904 Ingres 2006 00910 Ingres 2006 Release 2 00920 Ingres 9.2 00000 DBMS does not support SQL
INGRES/QUEL_LEVEL	Version of QUEL supported by the DBMS Examples: 00600 6.0 00601 6.1 00602 6.2 00603 6.3 00604 6.4 00605 OpenIngres1.x 00800 OpenIngres 2.0 and Ingres II 2.0 00850 Ingres II 2.5 00860 Ingres 2.6 00902 Ingres r3 00904 Ingres 2006 00910 Ingres 2006 Release 2 00920 Ingres 9.2 00000 DBMS does not support QUEL
INGRES_RULES	Y if rules supported N if rules not supported
INGRES_UDT	Y if user-defined data types supported N if user-defined data types not supported
INGRES_AUTH_GROUP	Y if group identifiers supported N if group identifiers not supported
INGRES_AUTH_ROLE	Y if role identifiers supported N if role identifiers not supported
INGRES_LOGICAL_KEY	Y if logical keys supported N if logical keys not supported
MAX_COLUMNS	Maximum number of columns allowed in a table. Current setting is 1024.
MIXEDCASE_NAMES	Y if case is significant in object names. N if ABC, Abc, and abc are all equivalent object names.
NATIONAL_CHARACTER_SET	Y if Unicode supported N if Unicode not supported
OPEN_SQL_DATES	Contains LEVEL 1 if the Enterprise

Capability	Value
	Access Server supports the OpenSQL date data type. Absent if OpenSQL date data type is implicitly supported when accessing a standard DBMS server.
OPEN/SQL_LEVEL	Version of OpenSQL supported by the DBMS Examples: 00600 6.0 00601 6.1 00602 6.2 00603 6.3 00604 6.4 00605 OpenIngres1.x 00800 OpenIngres 2.0 and Ingres II 2.0 00850 Ingres II 2.5 00860 Ingres 2.6 00902 Ingres r3 00904 Ingres 2006 Current setting is 00904. **Note:** Use this name instead of the deprecated COMMON/SQL_LEVEL.
OWNER_NAME	schema.table format is supported with with optional quotes. The default is QUOTED.
PHYSICAL_SOURCE	T indicates that iitables contains physical table information. P (a deprecated setting) indicates that only iiphysical_tables contains the physical table information. T is the default and only current usage.
QUEL_LEVEL	Text version of QUEL support level. Currently II9.2.0
SAVEPOINTS	Y if savepoints behave exactly as in Ingres (default) N if not
SLAVE2PC	Indicates if the DBMS supports Ingres 2-phase commit slave protocol: Y for Release 6.3 and above N for Star N usually for Enterprise Access

Capability	Value
	If not present, Y is assumed.
SQL_MAX_NCHAR_COLUMN_LEN	Maximum number of characters for an NCHAR column - 16000
SQL_MAX_NVCHR_COLUMN_LEN	Maximum number of characters for an NVARCHAR column - 16000
SQL_LEVEL	Text version of SQL support level. Currently II9.2.0
STANDARD_CATALOG_LEVEL	Release of the standard catalog interface supported by this database. Valid values: 00602 00604 00605 00800 00850 00860 00902 00904 00920 (the current setting)
UNIQUE_KEY_REQ	Y if the database service requires that some or all tables have a unique key. N or not present if the database service allows tables without unique keys.
SQL_MAX_BYTE_COLUMN_LEN	Maximum number of characters for a BYTE column - 32000
SQL_MAX_BYTE_LITERAL_LEN	Maximum number of characters for a BYTE LITERAL column - 32000
SQL_MAX_CHAR_COLUMN_LEN	Maximum number of characters for a CHAR column - 32000
SQL_MAX_CHAR_LITERAL_LEN	Maximum number of characters for a CHAR LITERAL column - 32000
SQL_MAX_COLUMN_NAME_LEN	Maximum number of characters for a column name - 32
SQL_MAX_ROW_LEN	Maximum length of a row - 262144
SQL_MAX_SCHEMA_NAME_LEN	Maximum number of characters for a schema name - 32
SQL_MAX_STATEMENTS	Maximum number of SQL statements If 0 unlimited

Capability	Value
SQL_MAX_TABLE_NAME_LEN	Maximum number of characters for a table name - 32
SQL_MAX_USER_NAME_LEN	Maximum number of characters for a user name - 32
SQL_MAX_VBYT_COLUMN_LEN	Maximum number of characters for a VARBYTE column - 32000
SQL_MAX_VCHR_COLUMN_LEN	Maximum number of characters for a VARCHAR column - 32000

iidbconstants Catalog

The iidbconstants catalog contains values required by the Ingres tools.

Column Name	Data Type	Description
user_name	char(32)	The name of the current user
dba_name	char(32)	The name of the database owner
system_owner	varchar(32)	The name of the catalog owner ($ingres)

iidistcols Catalog

The iidistcols catalog describes the columns that generate partitioning values for a partitioned table. Each partitioned table has one row per partitioning column per dimension in iidistcols. (Dimensions that do not use a value-based partitioning scheme do not appear in iidistcols.)

Column Name	Data Type	Description
table_name	char(32)	The name of the partitioned table
table_owner	char(32)	The owner of the table
dimension	smallint	The dimension being described, counting from 1
column_name	char(32)	The name of the partitioning column
column_sequence	smallint	The sequence of this column in this dimension's partitioning value, counting from 1
column_datatype	char(32)	The data type of the column: INTEGER SMALLINT

Column Name	Data Type	Description
		INT FLOAT REAL DECIMAL DOUBLE PRECISION CHAR CHARACTER VARCHAR LONG VARCHAR BYTE LONG BYTE C TEXT DATE MONEY

iidistschemes Catalog

The iidistschemes catalog describes the partitioning scheme of a partitioned table. Each partitioned table has one row per partitioning dimension in iidistschemes.

Column Name	Data Type	Description
table_name	char(32)	The name of the partitioned table
table_owner	char(32)	The owner of the table
dimension	smallint	The dimension being described, counting from 1
partitioning_columns	smallint	The number of columns that make up the partitioning value for a value-based partitioning rule
logical_partitions	smallint	The number of logical partitions in this dimension
partitioning_rule	varchar(9)	The partitioning rule: AUTOMATIC HASH LIST RANGE

iievents Catalog

The iievents catalog provides information about database events.

Column Name	Data Type	Description
event_name	char(32)	The name of the event
event_owner	char(32)	The owner of the event
text_sequence	integer8	The sequence number from 1 for the text_segment
text_segment	varchar(240)	The dbevent text definition
security_label	char(8)	Empty string This column is deprecated.

iifile_info Catalog

The iifile_info catalog holds the file name for a table or index. One row is returned for each location on which the table resides.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
owner_name	char(32)	The owner of the table
file_name	char(8)	Name of the file that contains the table
file_ext	char(3)	Extension of the file that contains an extent of the table. The first extent is named t00, succeeding extensions are named t01, t02, and so on.
location	char(32)	The location of the file
base_id	integer	Reltid from iirelation
index_id	integer	Reltidx from iirelation

iihistograms Catalog

The iihistograms table contains histogram information.

Column Name	Data Type	Description
table_name	char(32)	The table for the histogram

Column Name	Data Type	Description
table_owner	char(32)	The name of the owner
column_name	char(32)	The name of the column
text_sequence	integer8	The sequence number from 1 for the text_segment
text_segment	char(228)	The encoded histogram data created by optimizedb

iiindex_columns Catalog

For indexes, any columns that are defined as part of the primary index key has an entry in iiindex_columns. For a full list of all columns in the index, use the iicolumns catalog.

Column Name	Data Type	Description
index_name	char(32)	The index containing column_name
index_owner	char(32)	The name of the index owner
column_name	char(32)	The name of the column
key_sequence	smallint	Sequence of column in the key, numbered from 1
sort_direction	char(1)	Defaults to A for ascending

iiindexes Catalog

Each table with a table_type of I in the iitables table has an entry in iiindexes:

Column Name	Data Type	Description
index_name	char(32)	The index name
index_owner	char(32)	The name of the index owner
create_date	char(25)	Creation date of index
base_name	char(32)	The base table name
base_owner	char(32)	The base table owner
storage_structure	char(16)	The storage structure for the index: HASH ISAM BTREE

Column Name	Data Type	Description
		RTREE
is_compressed	char(1)	Y if the table is stored in compressed format N if the table is uncompressed Blank if unknown
key_is_compressed	char(1)	Y if the table uses key compression N if no key compression Blank if unknown
unique_rule	char(1)	U if the index is unique D if duplicate key values are allowed Blank if unknown
unique_scope	char(1)	R if this object is row-level S if statement-level Blank if not applicable
system_use	char(1)	S if the object is a system object U if user object G if generated by the system for the user Blank if unknown Used by utilities to determine which tables need reloading
persistent	char(1)	Y if the index re-created after a modify of the table N if not
index_pagesize	integer	The page size of an index

iiingres_tables Catalog

The iiingres_table catalog presents information about tables, views, and indexes in a different format than iitables.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The owner of the table
expire_date	char(25)	How long to save this table A value of 1970_01_01 00:00:00 GMT indicates table never expires.
table_integrities	char(1)	Y if integrities exist on this table N if not

Column Name	Data Type	Description
table_permits	char(1)	Y if permits exist on this table N if not
all_to_all	char(1)	Y if any user can perform any operation on this table N if not
ret_to_all	char(1)	Y if any user can retrieve data from this table
row_width	integer	Maximum width of tuple in bytes
is_journaled	char(1)	N if not journaled Y if journaled. C if journaled started/stopped after next checkpoint
view_base	char(1)	N if a view never existed on this table Y if at least one view existed for this table or all views on this table are dropped
modify_date	char(25)	Date of last modify performed on the table If never modified, the table creation date
table_ifillpct	smallint	Fill factor for B-tree index pages Otherwise unused
table_dfillpct	smallint	Fill factor for data pages if table does not have HEAP structure
table_lfillfct	smallint	Fill factor for B-tree leaf pages
table_minpages	integer	Minimum number of hash buckets to use if modifying to HASH structure
table_maxpages	integer	Maximum number of hash buckets to use if modifying to HASH structure
location_name	char(32)	Name of first location for data files
table_reltid	integer	Reltid from iirelation
table_reltidx	integer	Reltidx from iirelation

iiintegrities Catalog

The iiintegrities catalog contains one or more entries for each integrity defined on a table.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The owner of the table
create_date	char(25)	The creation date of the integrity
integrity_number	smallint	The number of the integrity
text_sequence	integer8	The sequence number from 1 for the text_segment
text_segment	varchar(240)	The text of the integrity definition

iikeys Catalog

The iikeys catalog contains information about keys used in internal indexes to support unique constraints and referential integrities.

Column Name	Data Type	Description
constraint_name	char(32)	The name of the constraint
schema_name	char(32)	The name of the schema
table_name	char(32)	The name of the table
column_name	char(32)	The name of the column
key_position	smallint	A number indicating the key position

iikey_columns Catalog

The iikey_columns catalog presents information about the key columns for indexes and base tables not using a heap structure.

Column Name	Data Type	Description
table_name	char(32)	The name of the table key is on
table_owner	char(32)	The owner of the table
column_name	char(32)	Name of key component column
key_sequence	smallint	Position of column in key. 1 being the most significant component

Column Name	Data Type	Description
sort_direction	varchar(1)	A: Ascending sort. (Currently only ascending indexes are supported.)

iilog_help Catalog

The iilog_help catalog presents information about table/view/index attributes (columns) in an alternate format to iicolumns.

Column Name	Data Type	Description
table_name	char(32)	Name of the object column is part of
table_owner	char(32)	The owner of the object
create_date	char(25)	Date object was created
table_type	char(8)	T if attribute is part of a table V if attribute is part of a view I if attribute is part of an index
table_subtype	char(1)	Always N
table_version	char(5)	II9.0 for current release of product
system_use	char(1)	S if part of a system catalog U if part of a user object
column_name	char(32)	Name of attribute.
column_datatype	char(32)	Long name of data type for this column
column_length	integer	Size in bytes of data
column_nulls	char(1)	N if not nullable Y if column supports nulls
column_defaults	char(1)	N if no default for this column Y if a default value exists for this column
column_sequence	smallint	Position of this column in table
key_sequence	smallint	Position in key for this table or zero

iilpartitions Catalog

The iilpartitions catalog describes each logical partition, and the partitioning values or range associated with that partition. Each logical partition of a partitioned table has at least one row in iilpartitions. Specifically, there is one row per column component for each partitioning value and for each logical partition in each dimension of the partitioned table.

Column Name	Data Type	Description
table_name	char(32)	The name of the partitioned table
table_owner	char(32)	The owner of the table
dimension	smallint	The dimension being described, counting from 1
logical_partseq	smallint	The logical partition sequence number in its dimension, counting from 1
partition_name	char(32)	The name of the partition If no name is assigned in the partition definition, a name of the form iipartNN is used, where NN is a sequence number.
value_sequence	smallint	The partitioning value being described: RANGE then incremental from 1 LIST then incremental from 1 AUTOMATIC then one entry with a zero value_sequence HASH then one entry with a zero value_sequence
column_sequence	smallint	The column component in the partitioning value: RANGE then incremental from 1 LIST then incremental from 1 AUTOMATIC then one entry with a zero column_sequence HASH then one entry with a zero column_sequence
operator	varchar(7)	If the partitioning is based on: RANGE then <, <=, =, >=, > LIST then =, DEFAULT AUTOMATIC then blank HASH then blank

Column Name	Data Type	Description
value	varchar (1500)	If the partitioning is based on: RANGE then column value LIST then column value (if DEFAULT, then meaningless) AUTOMATIC then NULL HASH then NULL

Here is an example of using iilpartitions to view the partitioning values for a table:

```
select dimension,
       logical_partseq
       value_sequence
       column_sequence
       operator varchar(value,30)
from iilpartitions
where table_name = 'partitioned_table'
and table_owner = 'thedba'
order by dimension, logical_partseq, value_sequence, column_sequence;
```

iimulti_locations Catalog

For tables located on multiple volumes, this table contains an entry for each additional location on which a table resides. The first location for a table can be found in the iitables catalog.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The owner of the table
loc_sequence	integer	The sequence of this location in the list of locations specified in the modify command. Numbered from 1.
location_name	char(32)	The name of the location

iipermits Catalog

The iipermits catalog contains one or more entries for each permit defined against a table, view, or procedure.

Column Name	Data Type	Description
object_name	char(32)	The name of the table, view, or procedure

Column Name	Data Type	Description
object_owner	char(32)	The owner of the table, view, or procedure
permit_grantor	char(32)	The name of the user granting the permit
object_type	char(1)	The type of the object: T if a table P if a database procedure E if an event V if a view
create_date	char(25)	The creation date of the permit
permit_user	char(32)	The user name to which this permit applies
permit_depth	smallint	Indicates relative ordering distance of the permit holder from the object owner, as established in the grant with grant option statements
permit_number	smallint	The number of this permit
text_sequence	integer8	The sequence number from 1 for the text_segment
text-segment	varchar(240)	The text of the permission definition

iiphysical_tables Catalog

Caution! The iiphysical_tables catalog will not exist in the next major release. Applications should query iitables for physical table information.

The information in the iiphysical_tables catalog overlaps with some of the information in iitables. This information is provided as a separate catalog primarily for use by Enterprise Access products. You can query the physical_source column, in iidbcapabilities, to determine whether you must query iiphysical_tables. If you do not want to query iidbcapabilities, you must always query iiphysical_tables to be sure of getting the correct information.

If a queriable object is type T or I (index Ingres installation only), it is a physical table and can have an entry in iiphysical_tables as well as iitables.

In most Enterprise Access products, this table is keyed on table_name plus table_owner.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The owner of the table
table_stats	char(1)	Y if this object has entries in the iistats table
table_indexes	char(1)	Y if this object has entries in iiindexes that see this as a base table
is_readonly	char(1)	Y if updates are physically allowed on this object
concurrent_access	char(1)	Y if concurrent access is allowed
num_rows	integer	The estimated number of rows in the table -1 if unknown
storage_structure	char(16)	The storage structure of the table: HEAP BTREE ISAM HASH
is_compressed	char(1)	Y if the table is compressed N if not compressed Blank if unknown
key_is_compressed	char(1)	Y if the table uses key compression N if no key compression

Column Name	Data Type	Description
		Blank if unknown
duplicate_rows	char(1)	U if rows must be unique D if duplicates are allowed Blank if unknown
unique_rule	char(1)	U if the storage structure is unique D if duplicates are allowed Blank if unknown or inapplicable
number_pages	integer	The estimated number of physical pages -1 if unknown
overflow_pages	integer	The estimated number of overflow pages -1 if unknown
row_width	integer	The size in bytes of the uncompressed binary value for the row -1 if unknown
allocation_size	integer	The allocation size, in pages -1 if unknown
extend_size	integer	The extend size, in pages -1 if unknown
allocated_pages	integer	The total number of pages allocated to the table
row_security_audit	char(1)	Y if per-row security auditing is enabled N if not
table_pagesize	integer	The page size of a table
table_relversion	smallint	Table layout version. Starts at zero when table is first created and is incremented when column layouts are altered; reset to zero when the table is modified.
table_reltotwid	integer	Width of table record in bytes
label_granularity	char(1)	An empty string This column is deprecated.
security_label	char(8)	An empty string This column is deprecated.

iiprocedures Catalog

The iiprocedures catalog contains one or more entries for each database procedure defined on a database.

Column Name	Data Type	Description
procedure_name	char(32)	The name of the procedure
procedure_owner	char(32)	The owner of the procedure
create_date	char(25)	The creation date of the procedure
proc_subtype	varchar(1)	N if native
text_sequence	integer	The sequence number from 1 for the test_segment
text_segment	varchar(240)	The text of the procedure definition
system_use	char(1)	U if the object is a user object G if generated by the system for the user
security_label	char(8)	An empty string This column is deprecated.
row_proc	char(1)	Y if the procedure is row producing N if the procedure is not row producing

iiproc_access Catalog

The iiproc_access catalog contains information about database procedures.

Column Name	Data Type	Description
object_name	char(32)	Name of database procedure
object_owner	char(32)	Owner of database procedure
permit_grantor	char(32)	Grantor of privilege to this procedure
object_type	char(1)	Always P (database procedure)
create_date	char(25)	Procedure creation date
permit_user	char(32)	Name of the grantee
permit_depth	smallint	Depth of dependencies this procedure permission depends on
permit_number	smallint	Reserved for future usage

Column Name	Data Type	Description
text_sequence	integer8	Sequence number from 1 for the text segment
text_segment	varchar(240)	The text of the procedure definition

iiproc_params Catalog

The iiproc_params catalog contains information about procedure parameters.

Column Name	Data Type	Description
procedure_name	char(32)	Name of database procedure
procedure_owner	char(32)	Owner of database procedure
param_name	char(32)	Name of parameter
param_sequence	smallint	Which argument this parameter corresponds to (1 = first)
param_datatype	char(32)	Data type of parameter
param_datatype_code	smallint	Numeric representation of datatype. See column_ing_datatype in iicolumns for these values.
param_length	integer	The column length Displays the precision for decimal data type, zero for money and date
param_scale	integer	Displays the scale for decimal data type, zero for all other data types
param_nulls	char(1)	Y if this parameter is NULLable
param_defaults	char(1)	Y if this parameter has a default value
param_default_val	varchar(1501)	Default value used if default parameter provided

iiproc_rescols Catalog

The iiproc_rescols catalog is a standard interface catalog with information about the parameters and result columns of an Ingres database procedure. It has one row for each parameter, the same as the rows of iiproc_params. It also contains one row for each result column of a row producing procedure.

Column Name	**Data Type**	**Description**
procedure_name	char(32)	The name of the procedure
procedure_owner	char(32)	The owner of the procedure
rescol_name	char(32)	The name of the parameter/result column
rescol_sequence	integer	Ordinal position of parameter or result column in procedure declaration
rescol_datatype	char(32)	Datatype of parameter/result column
rescol_datatype_code	smallint	Numeric representation of datatype
rescol_length	integer	Length of parameter/result column
rescol_scale	integer	The second number in a two-part user length specification; for type name (len1, len2) it is len2
rescol_nulls	char(1)	Y indicates this parameter is null
rescol_param	char(1)	Y indicates this is a parameter N indicates this is a result column
rescol_defaults	char(1)	Y indicates this parameter has a default value
rescol_default_val	varchar(1501)	Default value used if default parameter provided

iirange Catalog

The iirange catalog contains the range values for an rtree index.

Column Name	Data Type	Description
rng_baseid	integer	Identifier for the base table
rng_indexid	integer	Identifier for the rtree index table
rng_ll1	float8	Lower-left coordinate of range box for the first dimension
rng_ll2	float8	Lower-left coordinate of range box for the second dimension
rng_ll3	float8	Lower-left coordinate of range box for the third dimension. This column is currently not in use.
rng_ll4	float8	Lower-left coordinate of range box for the forth dimension. This column is currently not in use.
rng_ur1	float8	Upper-right coordinate of range box for the first dimension
rng_ur2	float8	Upper-right coordinate of range box for the second dimension
rng_ur3	float8	Upper-right coordinate of range box for the third dimension. This column is currently not in use.
rng_ur4	float8	Upper-right coordinate of range box for the forth dimension. This column is currently not in use.
rng_dimension	smallint	Dimension of range box Currently, the value is 2.
rng_hilbertsize	smallint	The size of the hilbert function for the range
rng_rangedt	smallint	The data type of the range box, either box or ibox
rng_rangetype	char(1)	The data type of the range box's coordinates: I if integer F if float

iiref_constraints Catalog

The iiref_constraints catalog contains information about referential constraints.

Column Name	Data Type	Description
ref_constraint_name	char(32)	The name of the referential constraint
ref_schema_name	char(32)	The name of the schema on which the referential constraint applies
ref_table_name	char(32)	The name of the table on which the referential constraint applies
unique_constraint_name	char(32)	The name of the unique constraint
unique_schema_name	char(32)	The name of the schema on which the unique constraint applies
unique_table_name	char(32)	The name of the table on which the unique constraint applies

iiregistrations Catalog

The iiregistrations catalog contains the text of register statements used by Star and Enterprise Access products.

Column Name	Data Type	Description
object_name	char(32)	The name of the registered table, view, or index
object_owner	char(32)	The name of the owner of the table, view, or index
object_dml	char(1)	The language used in the registration statement S if SQL Q if QUEL
object_type	char(2)	Object type: T if object is a table V if a view I if an index

Column Name	Data Type	Description
object_subtype	char(1)	Describes the type of table or view created by the register statement: L if this is a link for Star I if this is an imported object for Enterprise Access
text_sequence	integer8	The sequence number from 1 for the text_segment
text_segment	varchar (240)	The text of the register statement

iirules Catalog

The iirules catalog contains one row for each rule defined in a database.

Column Name	Data Type	Description
rule_name	char(32)	The name of the rule
rule_owner	char(32)	The name of the person who defined the rule
table_name	char(32)	The name of the table that the rule was defined against
text_sequence	integer8	The sequence number for the text segment
text_segment	varchar(240)	The text of the rule definition
system_use	char(1)	U if the object is a user object G if generated by the system for the user; used for constraints or views with check option

iisecurity_alarms Catalog

The iisecurity_alarms catalog contains information about the security alarms created on tables in the database. This catalog is a view of security alarm information held in the system iiprotect table.

Column Name	Data Type	Description
alarm_name	char(32)	The name of the security alarm
object_name	char(32)	The name of the table to which the security alarm applies

Column Name	Data Type	Description
object_owner	char(32)	The owner of the security alarm
object_type	char(1)	The type of object to which the security alarm applies Always T
create_date	char(25)	The date the security alarm was created
subject_type	char(1)	U if the security_user is a user G if a group R if a role P if a public identifier
security_user	char(32)	The user to which the security alarm applies
security_number	smallint	The security alarm number
dbevent_name	char(32)	Database event associated with the alarm
dbevent_owner	char(32)	Owner of the database event
dbevent_text	char(256)	Text of the database event
text_sequence	integer8	The sequence number from 1 for the text_segment
text_segment	varchar(24 0)	The text of the security alarm statement definition

iisession_privileges Catalog

The iisession_privileges catalog contains information about subject privilege statuses for the current session.

Column Name	Data Type	Description
priv_name	char(32)	The name of privilege
priv_access	char(1)	Y if privilege held N if privilege not held

iisequences Catalog

The iisequences catalog contains information about all sequences defined in the database.

Column Name	Data Type	Description
Seq_name	char(32)	The name of the sequence
Seq_owner	char(32)	The owner of the sequence
Create_date	ingresdate	The date on which the sequence was created
Modify_date	ingresdate	The date on which the sequence was last altered
Data_type	varchar(7)	The data type of the sequence integer bigint decimal
Seq_length	smallint	The size in bytes of the sequence value
Seq_precision	integer	The precision in decimal digits of the sequence value
Start_value	decimal(31)	The start value (or restart value) of the sequence
Increment_value	decimal(31)	The increment value of the sequence
Next_value	decimal(31)	The next sequence value to be assigned
Min_value	decimal(31)	The minimum value of the sequence
Max_value	decimal(31)	The maximum value of the sequence
Cache_size	integer	The number of cached sequence values
Start_flag	char(1)	Y if start value was defined N if start value was not defined
Incr_flag	char(1)	Y if increment value was defined N if increment value was not defined
Min_flag	char(1)	Y if minimum value was defined N if minimum value was not defined
Max_flag	char(1)	Y if maximum value was defined N if maximum value was not defined
Restart_flag	char(1)	Y if restart value was defined

Column Name	Data Type	Description
		N if restart value was not defined
Cache_flag	char(1)	Y if cache value was defined N if cache value was not defined
Cycle_flag	char(1)	Y if cycle was defined N if cycle was not defined
Order_flag	char(1)	Y if order was defined N if order was not defined
Unordered_flag	char(1)	Y if unordered sequence N if not unordered sequence
Seql_flag	char(1)	Y if sequential (not unordered) sequence N if not sequential sequence
Ident_flag	char(1)	Y if sequence associated with an identity column N if not

iistats Catalog

Th iistats catalog contains entries for columns that have statistics.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The owner of the table
column_name	char(32)	The column name to which the statistics apply
create_date	char(25)	The date on which statistics were gathered
num_unique	float4	The number of unique values in the column
rept_factor	float4	The repetition factor
has_unique	char(1)	Y if the column has unique values N if the column is not unique
pct_nulls	float4	The percentage (fraction of 1.0) of the table that contains NULL for the column
num_cells	smallint	The number of cells in the histogram
column_domain	smallint	A user-specified number signifying

Column Name	Data Type	Description
		the domain from which the column draws its values; default is 0
is_complete	char(1)	Y if the column contains all possible values in the domain N if the column does not contain all possible values in the domain
stat_version	char(8)	The version of the statistics for this column, for example, II9.0
hist_data_length	smallint	The length of the histogram boundary values: Either the specified length Or length computed by optimizedb

iisynonyms Catalog

The iisynonyms catalog contains information about the synonyms.

Column Name	Data Type	Description
synonym_name	char(32)	The name of the synonym
synonym_owner	char(32)	The owner of the synonym
table_name	char(32)	The name of the table, view or index for which the synonym was created
table_owner	char(32)	The owner of the table, view, or index for which the synonym was created

iitables Catalog

The iitables catalog contains an entry for each table, view, or index in the database.

Column Name	Data Type	Description
table_name	char(32)	The name of the table
table_owner	char(32)	The owner of the table
create_date	char(25)	The creation date of the object Blank if unknown
alter_date	char(25)	The last time this table was altered. Updated when the structure of the table changes through changes to the

Column Name	Data Type	Description
		columns in the table or to the primary key. Physical changes to the table, such as changes to data, secondary indexes, or physical keys, do not change this date. Blank if unknown.
table_type	char(1)	Type of the query object: T if table V if view I if index P if physical partition of a partitioned table Further information about views can be found in iiviews.
table_subtype	char(1)	Specifies the type of table or view. N if native for standard Ingres databases L if links for Star I if imported tables for Enterprise Access Blank if unknown
table_version	char(5)	Version of the object; enables the Ingres tools to determine where additional information about this particular object is stored. This reflects the database type, as well as the version of an object in a given database. For Ingres tables, the value for this field is II9.0.
system_use	char(1)	S if the object is a system object U if user object G if generated by the system for the user Blank if unknown
tups_per_page	integer	Maximum tuples per data page
keys_per_page	integer	Maximum keys per index page for ISAM and BTREE tables
keys_per_leaf	integer	Maximum keys per leaf for BTREE tables

The following columns have values only if the table_type is T, I, or P. Enterprise Access products that do not supply this information:

- Numeric columns are set to -1
- Character columns are set to blank.

Column Name	Data Type	Description
table_stats	char(1)	Y if the iistats table has entries N if the iistats table does not have entries Blank if query iistats to determine if statistics exist.
table_indexes	char(1)	Y if this object has entries in iiindexes that see this as a base table N if this object does not have entries Blank if query iiindexes on the base_table column
is_readonly	char(1)	N if updates are allowed Y if no updates are allowed Blank if unknown Used for tables defined to Enterprise Access for retrieval only (such as tables in a hierarchical database). If Y updates cannot occur, irrespective of the permissions set If N updates are allowed depending on the permissions setting
concurrent_access	char(1)	Y if concurrent access is allowed
num_rows	integer	The estimated number of rows in the table -1 if unknown If value is for a partitioned table, this is the total for all partitions
storage_structure	char(16)	The storage structure of the table: HEAP HASH BTREE ISAM
is_compressed	char(1)	Y if the table is compressed N if the table is uncompressed Blank if unknown

Column Name	Data Type	Description
key_is_compressed	char(1)	Y if the table uses key compression N if no key compression Blank if unknown
duplicate_rows	char(1)	D if the table allows duplicate rows U if the table does not allow duplicate rows Blank if unknown The table storage structure (unique vs. non-unique keys) can override this setting.
unique_rule	char(1)	D if duplicate keys are allowed. (A unique alternate key exists in iialt_columns and any storage structure keys are listed in iicolumns.) U if the object is an Ingres object, indicates that the object has unique storage structure keys. If the object is not an Ingres object, it indicates that the object has a unique key, described in either iicolumns or iialt_columns. Blank if uniqueness is unknown or does not apply.
number_pages	integer	The estimated number of used pages in the table -1 if unknown If the value is for a partitioned table, this is the total for all partitions.
overflow_pages	integer	The estimated number of overflow pages in the table -1 if unknown
partition_dimensions	smalliint	For a partitioned table, this is the number of dimensions (partitioning levels) in the table's partitioning scheme. In all other cases, this is zero.
phys_partitions	smallint	For a partitioned table, this is the number of physical partitions. For a physical partition, this is the partition number

Column Name	Data Type	Description
		In all other cases, this is zero.
row_width	integer	The size in bytes of the uncompressed binary value for a row of this query object

The following columns are used by the DBMS Server, except for those preceded by an asterisk (*).

Columns preceded by an asterisk (*) have values only if table_type is T or I.

Where Enterprise Access entries do not supply this information:

- Numeric columns are set to -1
- Character columns are set to blank

Column Name	Data Type	Description
expire_date	integer	Expiration date of table
modify_date	char(25)	The date when the table was last modified Blank if unknown or inapplicable
location_name	char(32)	The first location of the table. If there are additional locations for a table, they are shown in the iimulti_locations table and multi_locations are set to Y.
table_integrities	char(1)	Y if this object has Ingres style integrities Blank if query the iiintegrities table to determine if integrities exist
table_permits	char(1)	Y if this object has Ingres style permissions
all_to_all	char(1)	Y if this object has Ingres permit all to all N if not
ret_to_all	char(1)	Y if this object has Ingres permit retrieve to all N if not
is_journalled	char(1)	Y if journaling is enabled on this object N if journaling is disabled on this object

Column Name	Data Type	Description
		C if journaling is enabled/disabled after the next online checkpoint
view_base	char(1)	Y if object is a base for a view definition N if not Blank if unknown
multi_locations	char(1)	Y if the table is in multiple locations N if the table is single location
table_ifillpct	smallint	Fill factor, expressed as a percentage, for the index pages Specified in modify command nonleaffill clause
table_dfillpct	smallint	Fill factor, expressed as a percentage, for the data pages Specified in modify command fillfactor clause
table_lfillpct	smallint	Fill factor, expressed as a percentage, for the leaf pages Specified in modify command leaffill clause
table_minpages	integer	Minpages parameter from the last execution of the modify command. Used for hash structures only.
table_maxpages	integer	Maxpages parameter from the last execution of the modify command. Used for hash structures only.
table_relstamp1	integer	High part of last create or modify timestamp for the table
table_relstamp2	integer	Low part of last create or modify timestamp for the table
table_reltid	integer	Reltid from iirelation
table_reltidx	integer	Reltidx from iirelation
* unique_scope	char(1)	R if this object is row-level S if statement-level Blank if not applicable
* allocation_size	integer	The allocation size, in pages. Set to -1 if unknown.
* extend_size	integer	The extend size, in pages -1 if unknown

Column Name	Data Type	Description
* allocated_pages	integer	The total number of pages allocated to the table
row_security_audit	char(1)	Y if row-level security auditing is enabled N if not
table_pagesize	integer	Page size of a table
table_relversion	smallint	Version of table
table_reltotwidth	integer	Width of the table, including all deleted columns
table_reltcpri	smallint	Indicates a table's priority in the buffer cache Values can be between 0 - 8: Zero is the default. 1 - 8 can be specified in the priority clause of a create table or modify table statement.
label_granularity	char(1)	Empty string This column is deprecated.
security_label	char(8)	Empty string This column is deprecated.

iiviews Catalog

The iiviews catalog contains one or more rows for each view in the database.

Column Name	Data Type	Description
table_name	char(32)	The name of the view
table_owner	char(32)	The owner of the view
view_dml	char(1)	The language in which the view was created: S if SQL Q if QUEL
check_option	char(1)	Y if the check option was specified N if not Blank if unknown
text_sequence	integer8	The sequence number from 1 for the text_segment

Column Name	Data Type	Description
text_segment	varchar(240)	The text of the view definition

Standard Catalogs for iidbdb

The master database (iidbdb) contains these additional Standard Catalogs:

iiaudit

iidatabase_info

iidbprivileges

iiextend_info

ii_location_info

ii_profiles

iirollgrants

iiroles

iisecurity_state

iiusers

iiaudit Catalog

The iiaudit catalog provides the information from which a user (with security privilege) can read the security audit log. This catalog is a read-only virtual representation of the underlying non-Ingres table.

For information on reading the audit log, see the *Security Guide*.

Column Name	Data Type	Description
audittime	date	The time when the security event occurred
user_name	char(32)	The effective name of the user that triggered the security event
real_name	char(32)	The real name of the user
userprivileges	char(32)	Privileges associated with the user session, with letters denoting the possession of a subject privilege
objprivileges	char(32)	Privileges granted to the user, with letters denoting the possession of a subject privilege
database	char(32)	The name of the database in which the event was triggered

Column Name	Data Type	Description
auditstatus	char(1)	Y—the attempted operation was successful N—the attempted operation was unsuccessful
auditevent	char(24)	The type of event: select insert delete update copy into copy from execute modify create destroy security
objecttype	char(24)	The type of object being accessed: database role procedure table location view security user security alarm rule event
objectname	char(32)	The name of the object being accessed
objectowner	char(32)	The owner of the object being accessed
description	char(80)	The text description of the event
sessionid	char(16)	The session associated with the event
detailinfo	char(256)	Detailed information about the event
detailnum	integer	The sequence number for multiple detail items needed to describe the event
querytext_sequence	integer	Identifier for associated prepared query

iidatabase_info Catalog

This catalog describes attributes for a database.

Column Name	Data Type	Description	
database_name	char(32)	Name of the database	
database_owner	char(32)	Owner of the database	
data_location	char(32)	Default data location for this database	
work_location	char(32)	Default work location for this database	
ckp_location	char(32)	Default checkpoint location for this database	
jnl_location	char(32)	Default journal location for this database	
dump_location	char(32)	Default dump file location for this database	
compat_level	char(4)	The compatibility level of the Ingres database Currently 9.04	
compat_level_minor	integer	Unused; defaults to 0	
database_service	integer	Database services available (such as: Is the database distributed? Can it be accessed through gateways?) Bitmask of database attributes:	
		0x00000000	Default
		0x00000001	Distributed database
		0x00000002	Coordinator database for a distributed database
		0x00000004	Gateway database
		0x00010000	Regular IDs are translated to upper case
		0x00020000	Database created with LP64
		0x00040000	Delimited IDs are translated to upper case
		0x00080000	Delimited IDs are not translated
		0x00100000	Real user IDs are not translated
		0x00200000	Unicode types in Normal Form C can be stored in database

Column Name	Data Type	Description	
		0x40000000	Database forced consistent by verifydb
		0x80000000	Unicode types in Normal Form D can be stored in database
security_label	char(8)	Empty string This column is deprecated.	
access	integer	Bitmask of database access attributes: Bitmasks as follows:	
		0x00000000	Database is private
		0x00000001	Database is globally accessible
		0x00000002	Unused
		0x00000004	Database was/is in process of being destroyed
		0x00000008	Database was/is in process of being created
		0x00000010	Database is operational, i.e. is accessible to users
		0x00000020	Database was created in an earlier Ingres release and has not yet been upgraded
		0x00000040	Database was created via an earlier Ingres version and is in the process of being upgraded or the upgrade attempt was made and failed
		0x00000080	Database created with B1 security
		0x00000100	Do not wait during destroydb if the database is busy
		0x00000200	Production mode
		0x00000400	No online checkpoints
		0x00000800	Database is read only
database_id	integer	Unique numeric identifier for this database in the	

Column Name	Data Type	Description
		installation

iidbprivileges Catalog

The iidbprivileges catalog contains information about the privileges defined in a database.

Column Name	Data Type	Description
database_name	char(32)	The name of the database on which the privilege is defined
grantee_name	char(32)	The name of the grantee for which the privilege is granted: User Group Role Public
gr_type	char(1)	Authorization type of the grantee: U—user G—group R—role P—public
cr_tab	char(1)	Indicates if the grantee has the create table privilege: U—undefined Y—yes N—no
cr_proc	char(1)	Indicates if the grantee has the create procedure privilege: U—undefined Y—yes N—no
lk_mode	char(1)	Indicates if the grantee has the set lockmode privilege: U—undefined Y—yes N—no
db_access	char(1)	Y if grantee can connect to databases
up_syscat	char(1)	Y if grantee can update catalog tables
db_admin	char(1)	Indicates if the grantee has the db_admin privilege: U—undefined

Column Name	Data Type	Description
		Y—yes N—no
global_usage	char(1)	Reserved for future use
qry_io_lim	integer	The limit of I/O per query for the grantee if qry_io is Y
qry_io	char(1)	Indicates whether the query_io_limit privilege has been defined for the database and authorization type specified in database_name and grantee_name, respectively: Y—limit exists N—no limit U—undefined
qry_row_lim	integer	The limit of query rows per query for the grantee if qry_row is Y
qry_row	char(1)	Indicates whether the query_row_limit privilege has been defined for the database and authorization type specified in database_name and grantee_name, respectively. Y—limit exists N—no limit U—undefined
sel_syscats	char(1)	Y if grantee has select_syscat privileges
tbl_stats	char(1)	Y if grantee has table_statistics privileges
idle_time	char(1)	Y if grantee has an idle time limit
idle_time_lim	integer	Idle time limit in seconds
conn_time	char(1)	Y if grantee has a connect time limit
conn_time_lim	integer	Connect time limit in seconds
sess_prio	char(1)	Y if grantee has the session priority privilege and can alter session priorities
sess_pri_lim	integer	Highest priority to which a session owned by this grantee can be set

iiextend_info Catalog

The iiextend_info catalog provides information about which locations databases have been extended to:

Column Name	Data Type	Description	
location_name	char(32)	Location name for this extent	
database_name	char(32)	Name of database extended to location_name	
status	integer	Status of this extent Bitmasks are as follows:	
		0x00000001	Database has been successfully extended to this location
		0x00000002	Location is used as a data location
		0x00000004	Location is used as a work location
		0x00000008	Location is used as a auxiliary work location
raw_start	integer	Default is 0	
raw_blocks	integer	Default is 0	

iilocation_info Catalog

The iilocation_info catalog contains information about the database locations.

Column Name	Data Type	Description
location_name	char(32)	The name of the location
data_usage	char(1)	Y if data location N if not
jrnl_usage	char(1)	Y if journal location N if not
ckpt_usage	char(1)	Y if checkpoint location N if not
work_usage	char(1)	Y if work location N if not
dump_usage	char(1)	Y if dump location N if not

Column Name	Data Type	Description	
awork_usage	char(1)	Y if auxiliary work location N if not	
location_area	char(128)	The location of the area, either: II_CHECKPOINT II_DATABASE II_WORK II_JOURNAL II_DUMP or directory name	
security_label	char(8)	Empty string This column is deprecated.	
raw_pct	integer	Percentage of the raw device allocated to this location	
status	integer	What the location is used for: Bitmasks as follows:	
		0x00000001	General purpose
		0x00000002	Dump
		0x00000008	Database
		0x00000010	Work
		0x00000020	Auxiliary work
		0x00000040	Journal
		0x00000200	Checkpoint

iiprofiles Catalog

Iiprofiles is the standard catalog interface to user profile information.

Column Name	Data Type	Description
profile_name	char(32)	Name of profile
createdb	char(1)	Y if profile gives by default the right to create databases R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right

Column Name	Data Type	Description
trace	char(1)	Y if profile gives by default the right to enabling usage of tracing and debugging features R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right
audit_all	char(1)	Y if security audit of all user activity is enabled by this profile N if profile does not give this right
security	char(1)	Y if profile gives by default the right to use security related functions such as the creation or deletion of users R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right
maintain_locations	char(1)	Y if profile gives by default the right to create and change the characteristics of database and file locations R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right
operator	char(1)	Y if profile gives by default the right to perform database maintenance operations R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right
maintain_users	char(1)	Y if profile gives by default the right to create, alter or drop users, profiles, groups, and roles, and to grant or revoke database and installation resource controls R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right
maintain_audit	char(1)	Y if profile gives by default the right to enable, disable, or alter security audit, and to change

Column Name	Data Type	Description	
		security audit privileges R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right	
auditor	char(1)	Y if profile gives by default the right to register, remove and query audit logs R if this subject privilege is enabled by this profile, but is not part of the default privileges for this profile N if profile does not give this right	
audit_query_text	char(1)	Y if security audit of query text is enabled for this profile N if profile does not give this right	
expire_date	date	Date when profile expires Blank if expiration date was not specified	
lim_sec_label	char(8)	Empty string	
default_group	char(32)	If specified, group to use if no explicit group was specified when accessing the database and user using this profile does not have an explicit default group, or nogroup specified	
internal_status	integer	Numeric representation of privileges associated with this profile Bitmasks as follows:	
		0x00000001	createdb
		0x00000004	trace
		0x00000200	operator
		0x00000400	audit_all
		0x00000800	maintain_locations
		0x00002000	auditor
		0x00004000	maintain_audit
		0x00008000	security
		0x00010000	maintain_users
		0x01000000	audit_security_text

iirollgrants Catalog

The standard catalog interface to information about role grants.

Column Name	Data Type	Description
roll_name	char(32)	Name of granted role
gr_type	char(1)	Type of grant: U—user G—group R—role P—public Blank
grantee_name	char(32)	Name of grantee
admin_option	char(1)	Y if grantee can GRANT this role to others N if not

iiroles Catalog

The standard catalog interface to information about role identifiers.

Column Name	Data Type	Description
role_name	char(32)	Name of this role
createdb	char(1)	Y if role provides right to create databases , N otherwise
trace	char(1)	Y if role enables usage of tracing and debugging features, N otherwise
audit_all	char(1)	Y if security audit of all user activity is enabled by this role, N otherwise
security	char(1)	Y if role allows usage of security-related functions such as the creation or deletion of users, N
maintain_locations	char(1)	Y if role allows the user to create and change the characteristics of database and file locations, N
operator	char(1)	Y if role allows the user to perform database maintenance operations, N
maintain_users	char(1)	Y if role enables the right to create, alter or drop users, profiles, groups, and roles, and to grant or revoke database and installation resource controls, N
maintain_audit	char(1)	Y if role allows user to enable, disable, or alter

Column Name	Data Type	Description	
		security audit, and to change security audit privileges, N	
auditor	char(1)	Y if role enables the registering, removing, and querying of audit logs, N	
audit_query_text	char(1)	Y if security audit of query text is enabled by this profile, N	
security	char(8)	Empty string	
lim_sec_label	char(8)	Empty string	
internal_status	integer	Numeric representation of privileges associated with this status. Number is a bitmask as follows:	
		0x00000001	createdb
		0x00000004	trace
		0x00000200	operator
		0x00000400	audit_all
		0x00000800	maintain_locations
		0x00002000	auditor
		0x00004000	maintain_audit
		0x00008000	security
		0x00010000	maintain_users
		0x01000000	audit_security_text
internal_flags	integer	Reserved for future use	

iisecurity_state Catalog

The iisecurity_state catalog contains information about the security auditing state of the Ingres installation.

Column Name	Data Type	Description
type	char(16)	Type of security audit activity: Event - security-relevant events Unknown
name	char(32)	The name of the security audit class: Alarm All

Column Name	Data Type	Description
		Database Dbevent Installation Location Procedure Query_text Resource Role Row Rule Security Table User View Unknown
state	char(1)	E if this security audit class is enabled D if disabled
number	integer	Unique identifier for this activity-type / audit class: 1 Database 2 Role 3 Procedure 4 Table 5 Location 6 View 7 Row 8 Security 9 User 11 Alarm 12 Rule 13 Dbevent 14 Installation 15 All 16 Resource 17 Query_text

iiusers Catalog

The iiusers catalog contains information about the privileges held by users.

Column Name	Data Type	Description
user_name	char(32)	The name of the user

Column Name	Data Type	Description
createdb	char(1)	Y if the user has the default right to create databases R if the user has the right but not by default N if the user does not have the right
trace	char(1)	Y if the user has the default right to use tracing and debugging features R if the user has the right but not by default N if the user does not have the right
audit_all	char(1)	Y if the user has the right to security audit all user activity N if the user does not have the right
security	char(1)	Y if the user has the default right to use security-related functions such as creating and deleting users R if the user has the right but not by default N if the user does not have the right
maintain_locations	char(1)	Y if the user has the default right to create and change the characteristics of database and file locations R if the user has the right but not by default N if the user does not have the right
operator	char(1)	Y if the user has the default right to perform database maintenance operations R if the user has the right but not by default N if the user does not have the right
maintain_users	char(1)	Y if the user has the default right to create, alter or drop users, profiles, groups, and roles, and to grant or revoke database and installation resource controls R if the user has the right but not by default N if the user does not have the right
maintain_audit	char(1)	Y if the user has the default right to enable, disable, or alter security audit, and to change security audit privileges R if the user has the right but not by default N if the user does not have the right

Column Name	Data Type	Description	
auditor	char(1)	Y if the user has the default right to register, remove, and query audit logs R if the user has the right but not by default N if the user does not have the right	
audit_query_text	char(1)	Y if the user can see query text, enabled if security_audit=(query_text) was specified when creating or altering the user N if the user does not have the right	
expire_date	date	Optional expiration date after which the user cannot log on	
profile_name	char(32)	The profile associated with this user or blank	
lim_sec_label	char(8)	Empty string	
default-group	char(32)	The user's default group or blank	
internal_status	integer	Numeric representation of privileges associated with this status. Number is a bitmask as follows:	
		0x00000001	createdb
		0x00000004	trace
		0x00000200	operator
		0x00000400	audit_all
		0x00000800	maintain_locations
		0x00002000	auditor
		0x00004000	maintain_audit
		0x00008000	security
		0x00010000	maintain_users
		0x01000000	audit_security_text
internal_def_priv	integer	Numeric representation of default privileges, bitmask as above	
internal_flags	integer	Numeric representation of Ingres system privileges held by the user	

Mandatory and Ingres-Only Standard Catalogs

Mandatory catalogs are those catalogs that are required for all installations, including Enterprise Access and non-Enterprise Access installations. Ingres-only catalogs are required for non-Enterprise Access installations.

Mandatory Catalogs With Entries Required

The following catalogs must be present on both Enterprise Access and non-Enterprise Access installations. These catalogs must contain entries.

- iicolumns
- iidbcapabilities
- iidbconstants
- iisynonyms
- iitables

Mandatory Catalogs Without Entries Required

The following catalogs must be present on both Enterprise Access and non-Enterprise Access installations. However, these catalogs are not required to contain entries.

- iialt_columns
- iiaudit
- iiaudit_tables
- iiconstraint_indexes
- iiconstraints
- iidb_comments
- iidb_subcomments
- iihistograms
- iiindex_columns
- iiindexes
- iikeys
- iiprocedures
- iiref_constraints
- iiregistrations
- iisecurity_alarms
- iistats
- iiviews

Ingres-Only Catalogs

The following catalogs are required by non-Enterprise Access Ingres installations.

- iifile_info
- iiintegrities
- iilog_help
- iimulti_locations
- iipermits
- iirules

Extended System Catalogs

Extended System catalogs are used by the Ingres tool products, such as ABF, VIFRED, and RBF, to store information on user interface objects such as applications, forms, and reports.

To add or upgrade the catalogs required by a product, you must use the upgradefe command.

Organization of Extended System Catalogs

For the purpose of installing and upgrading products, extended system catalogs are grouped into modules. Each product requires one or more of these modules. The contents of a module are subject to change with a new release, as catalogs are added or changed.

By default, when you create a database (using the createdb command) or upgrade a product (using the upgradefe command), catalogs are created for all products for which you are authorized. To create catalogs only for specific products, you must specify the products on the createdb command line. Valid products are:

- ingres (the base product)
- ingres/dbd
- vision
- windows/4gl

Each of these products requires one or more of the following modules:

Module Name	**Catalogs in Module**
APPLICATION_DEVELOPMENT_1 Base product catalogs Required by: Ingres Ingres Vision	ii_abfclasses ii_abfdependencies ii_abfobjects ii_encoded_forms ii_fields ii_forms ii_gropts ii_joindefs ii_qbfnames ii_rcommands ii_reports ii_sequence_values ii_trim
APPLICATION_DEVELOPMENT_2	ii_applications ii_components

Module Name	Catalogs in Module
Catalogs for OpenROAD Required by: OpenROAD	ii_dependencies ii_incl_apps ii_srcobj_encoded ii_stored_bitmaps ii_stored_strings
APPLICATION_DEVELOPMENT_3 Required by: Ingres Vision	ii_framevars ii_menuargs ii_vqjoins ii_vqtabcols ii_vqtables
CORE Catalogs for Ingres and Ingres tools base product Required by: All products	ii_app_cntns_comp ii_client_dep_mod ii_dict_modules ii_encodings ii_entities ii_id ii_locks ii_longremarks ii_objects
DATA_MODEL catalogs for distributed Ingres Required by: INGRES/DBD version 1	ii_atttype ii_atttype_version ii_defaults ii_databases ii_dbd_acl ii_dbd_identifiers ii_dbd_locations ii_dbd_rightslist ii_dbd_table_char ii_domains ii_enttype ii_joinspecs ii_key_info ii_key_map ii_limits ii_rel_cncts_ent ii_reltype ii_sqlatts ii_sqltables
METASCHEMA Required by: INGRES/DBD version 2	ii_atttype ii_defaults ii_databases ii_domains ii_enttype ii_joinspecs ii_key_info ii_key_map

Module Name	Catalogs in Module
	ii_limits ii_rel_cncts_ent ii_reltype ii_sqlatts ii_sqltables
PHYSICAL_DATA_MODELLING Required by INGRES/DBD version 2	ii_dbd_acl ii_dbd_identifiers ii_dbd_locations ii_dbd_rightslist ii_dbd_table_char

Data Dictionary Catalogs

Data dictionary catalogs contain the names of the catalogs installed for Ingres tools (such as ABF, QBF, and VIFRED). When you invoke one of these products, the product uses the data dictionary catalogs to determine if the required catalogs are present; if the catalogs are not present, the product cannot run.

The ii_client_dep_mod catalog lists the products that have been installed, and the modules required by the product. The format of ii_client_dep_mod is as follows:

Column Name	Data Type	Description
client_name	varchar(32)	Name of the product (for example, Ingres).
client_version	integer	Release of the product.
module_name	varchar(32)	Module required by the product.
module_version	integer	Release of module required by this release of the product.
short_remark	varchar(60)	Comment regarding product/module information.

The ii_dict_modules catalog lists the modules that are installed in the database. (A module is a group of catalogs.) The format of ii_dict_modules is as follows:

Column Name	Data Type	Description
module_name	varchar(32)	Name of the installed module (for example, CORE).

Column Name	Data Type	Description
module_version	integer	Release number of the installed module.
short_remark	varchar(60)	Comment about the module.

Object IDs in Extended System Catalogs

Every user interface object (form, ABF frame, ABF application, report, QBF JoinDef, and so on) is identified in the extended system catalogs by a unique number, the object ID. The object ID is generated when the object is created. For each database, the largest object ID issued to date is stored in the table ii_id; this value is incremented and issued as the ID for each new object.

An object's name, owner, and other information are stored once only, in the ii_objects catalog. In all other extended system catalogs, objects are identified by their object ID.

User programs that insert objects into the extended system catalogs must generate a unique object ID for each new object by incrementing the object_id column in the ii_id catalog. Be sure to keep the transaction that updates ii_id.object_id as short as possible and to recover properly from errors. For information on handling errors in transactions, see the *SQL Reference Guide*.

Copying the Extended System Catalogs

Extended system catalogs must only be copied into new databases, and never into existing databases that contain user interface objects (such as forms or reports).

Copying extended system catalogs with the copy statement does not create new object IDs for the copied objects. If the target database already contains user interface objects, serious problems can result: different objects with the same object ID (for example, both a form and a report with the same object ID). Use the appropriate copy utility (copyform, copyrep) to copy objects to existing databases; the copy utilities generate a new object ID for each object copied into the target database.

Catalogs Shared by All Ingres Tools

The following extended system catalogs are used by all Ingres tools:

- ii_encodings
- ii_id
- ii_locks
- ii_longremarks
- ii_objects

ii_encodings Catalog

The ii_encodings catalog contains 4GL frames and procedures encoded into a compact and portable form. Objects in this catalog are referred to as encoded objects.

This catalog is structured as btree unique on the encode_object and encode_sequence columns:

Column Name	Data Type	Description
object_id	integer	Currently not used. Is set to either 0 or the same value as the encode_object column.
encode_object	integer	The object ID for this encoded object. Various other information about this encoded object (such as name and owner) is kept in the ii_objects catalog.
encode_sequence	smallint	A sequence number, starting from 0, for the rows comprising a single encoded object. Because objects, for example a 4GL frame, can be arbitrarily long, an arbitrary number of ii_encodings rows are required to encode the object.
encode_estring	varchar (1990)	A segment of the encoded string.

ii_id Catalog

The ii_id catalog is a heap table containing one column with a single row. The value in this catalog is the highest object ID currently allocated in this database. For a newly created database, this value is initialized to 10000 and can grow as large as the largest positive integer value. Object IDs below 10000 are reserved for system use:

Column Name	Data Type	Description
object_id	integer	The highest current object ID in this database.

ii_locks Catalog

The ii_locks catalog is used by ABF, Vision, and OpenROAD to manage concurrent user access to applications and application components (such as frames or procedures). The format of ii_locks is as follows:

Column Name	Data Type	Description
entity_id	integer	Object ID of the locked object (application or application component).
session_id	integer	ID of the user session that locked the object.
locked_by	varchar(32)	User ID that locked the object.
lock_date	char(25)	Date locked.
lock_type	varchar(16)	Type of lock: write if an application component is locked. refresh if a concurrent application user has changed the application flow diagram (possibly affecting other users' screens). entry if no change to flow diagram.

ii_longremarks Catalog

The ii_longremarks catalog contains the "long remarks" text associated with user interface objects. Only those objects that have an associated long remark are entered in this catalog. Consequently, unless all objects being selected have a long remark entered, joins between ii_objects and ii_longremarks must be outer joins. For an example of an outer join between the ii_objects and ii_longremarks catalogs, see Sample Queries for the Extended System Catalogs for SQL (see page 495). The current implementation restricts long remarks to a single row; the sequence column is provided for a future enhancement to allow remarks of arbitrary length.

The ii_longremarks catalog is structured as btree unique on the object_id and remark_sequence columns:

Column Name	**Data Type**	**Description**
object_id	integer	Object ID of the user interface object this remark belongs to. Various other information about this object (such as name, owner and object class) is kept in the ii_objects catalog.
remark_sequence	integer	A sequence number for (future) representation of multiple segments of text comprising one object's long remarks.
long_remark	varchar(600)	The long remarks text associated with the object.
remark_language	integer	Currently unused.

ii_objects Catalog

The ii_objects catalog contains a row for every user interface object in the database. This catalog stores basic information about each object, such as name, owner, object ID, object class, and creation date.

Objects in this table often have additional information represented in rows of one or more other user interface catalogs; for example, form objects are also represented by rows in ii_forms, ii_fields, and ii_trim. In all cases, the object ID is the key column that is used to join information from multiple catalogs about a single object.

The ii_objects catalog is a btree table, keyed on the object_class, object_owner, and object_name columns. The ii_objects catalog has a secondary index, btree unique, keyed on the object_id column:

Column Name	Data Type	Description
object_id	integer	The object identifier, unique among user interface objects in the database.
object_class	integer	The object's class. Tells what type of object this is (form, report, and so on). For object class definitions, see Object Classes in the ii_objects Catalog (see page 494).
object_name	varchar(32)	The name of the object.
object_owner	varchar(32)	The object owner's user name.
object_env	integer	Currently unused.
is_current	smallint	Currently unused.
short_remark	varchar(60)	A short descriptive remark associated with the object.
object_language	smallint	Currently unused.
create_date	char(25)	The time and date when the object was initially created.
alter_date	char(25)	The time and date when the object, or associated information, was most recently altered or saved.
alter_count	integer	A count of the number of times this object has been altered or saved.
last_altered_by	varchar(32)	The name of the user who last altered or saved this object.

Object Classes in the ii_objects Catalog

Object class is a column in the ii_objects catalog. Each object class is as follows:

Object Class	Description
1002	JoinDef
1501	Generic Report
1502	Report-Writer Report
1511	RBF Report
1601	Form
2001	ABF Application
2010	4GL Intermediate Language Code
2021	Host Language Procedure
2050	4GL Procedure
2075	Database Procedure
2110	Global Variable
2120	Constant
2130	Record Definition
2133	Record Attribute
2190	Undefined Procedure
2201	QBFName
2210	4GL Frame
2219	Old 4GL Frame
2220	Report Frame
2230	QBF Frame
2249	GBF Frame
2250	Undefined Frame
2260	Vision menuframe
2261	Vision append frame
2262	Vision update frame
2264	Vision browse frame
3001	ABF Form Reference

Object Class	Description
3501	Dependency Type: member of
3502	Dependency Type: database reference
3503	Dependency Type: call with no use of return code
3504	Dependency Type: call with use of return code

Sample Queries for the Extended System Catalogs for SQL

You can issue queries to get information from the extended system catalogs. Each query specifies the class code for the type of object being selected.

For details on class codes, see Object Classes in the ii_objects Catalog (see page 494).

Example: Find Information on Every Report in the Database

This query finds information on every report in the database.

```
select report=o.object_name, o.object_owner,
  o.short_remark, r.reptype
  from ii_objects o, ii_reports r
  where (o.object_class = 1501 or
    o.object_class = 1502 or
    o.object_class = 1511)
   /* object_classes 1501, 1502, 1511 = reports
   */
  and o.object_id = r.object_id
```

Example: Find the Name and Tabbing Sequence Number of Fields on a Form

This query finds the name and tabbing sequence number of every simple field and table field on form “empform” (empform is owned by user “susan”).

```
select form=o.object_name, f.fldname, f.flseq,
  f.fltype
  from ii_objects o, ii_fields f
  where o.object_class = 1601
  /* object_class 1601 = "form" */
  and o.object_name = 'empform'
  and o.object_owner = 'susan'
  and o.object_id = f.object_id
  and (f.fltype = 0 or f.fltype = 1)
  /* simple field or table field */
  order by flseq
```

Example: Find Information on Every ABF Application

This query finds information on every ABF application in the database.

```
select appname=object_name, object_owner
  from ii_objects o
  where o.object_class = 2001
  /* object_class 2001 = "abf application" */
```

Example: Find Information on All Frames and Procedures in an Application

The following two queries require two correlation variables on the table ii_objects. Two variables are required, because we need to find all the frames and procedures in the application, plus object information on the selected frames and procedures.

This query finds information on all frames and procedures in application lab.

```
select appname=o.object_name, o2.object_class,
 2.object_name, o2.object_owner, o2.short_remark
  from ii_objects o, ii_abfobjects a,
    ii_objects o2
  where o.object_name = 'lab'
  and o.object_class = 2001
  /* object_class 2001 = "abf application" */
  and o.object_id = a.applid
  and a.object_id = o2.object_id
```

This query finds dependency information for all frames and procedures in application payables. Frames and procedures with no dependencies show up as a row with ad.name=DUMMY.

```
select appname=o.object_name, o2.object_class,
 o2.object_name, o2.object_owner,
  o2.short_remark, ad.abfdef_name,
  ad.abfdef_deptype, ad.object_class
  from ii_objects o, ii_objects o2,
    ii_abfobjects a, ii_abfdependencies ad
  where o.object_name = 'payables'
  and o.object_class = 2001
/* object_class 2001 = "abf application" */
  and o.object_id = a.applid
  and a.object_id = o2.object_id
  and a.object_id = ad.object_id
order by object_name
```

Example: Select Object Information

This query selects object information and long remarks, when available, by performing an outer join of ii_objects with ii_longremarks.

```
select o.object_name, o.object_class,
  o.object_owner, o.short_remark, l.long_remark
  from ii_objects o, ii_longremarks l
  where o.object_id = l.object_i
  union all

select o.object_name, o.object_class,
  o.object_owner, o.short_remark, ''
  from ii_objects o
  where not exists
  ( select *
  from ii_longremarks
  where ii_longremarks.object_id = o.object_id )
order by object_name
```

Forms System Catalogs

The forms system requires the following extended system catalogs:

- ii_encoded_forms
- ii_fields
- ii_forms
- ii_trim

ii_encoded_forms Catalog

The ii_encoded_forms catalog contains encoded versions of forms. The encoding allows forms to be retrieved from the database faster.

The ii_encoded_forms catalog is structured as compressed btree unique on the object_id and cfseq columns:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying this form in the ii_objects catalog. Other information about this form (such as name, owner, and object class) is stored in the ii_objects catalog.
cfseq	smallint	Sequence number of this record for a

Column Name	Data Type	Description
		particular encoded form. Record sequence numbering starts at zero (0).
cfdatsize	integer	Number of bytes of actual data in column cfdata.
cftotdat	integer	Total number of bytes needed to hold an encoded form.
cfdata	varchar(1960)	Data area used for holding an encoded form.

ii_fields Catalog

The ii_fields catalog contains information on the fields in a form. For every form, there is one row in this catalog for each field, table field and table field column. As used below, the word field refers to a simple field, a table field or a column in a table field. An example of a query that selects information about fields on a form is in Querying the Extended System Catalogs for SQL (see page 495).

The ii_fields catalog is structured as btree unique on the object_id and flsubseq columns:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying the form this field belongs to in the ii_objects catalog. Other information about the form (such as name, owner, and object class) is stored in the ii_objects catalog.
flseq	smallint	The sequence number of the field in the form (or column in a table field). This determines the tabbing order among fields and among columns in a table field.
fldname	varchar(32)	The name of the field.
fldatatype	smallint	The field's data type. Possible values are listed below with nullable data types being the negative of the listed value: 3 date

Column Name	Data Type	Description
		5 money
		20 char
		21 varchar
		30 integer
		31 floating point
		32 c
		37 text
fllength	smallint	The internal data length of the field in bytes. This cannot be the same as the user-defined length. This is the length used by Ingres.
flprec	smallint	Reserved for future use.
flwidth	smallint	The number of characters displayed in the field on the form including wrap characters. For example, if the format for the field is c20.10, flwidth is 20. If the field is a table field, this value is the number of columns in the table field.
flmaxlin	smallint	The number of lines occupied by the field (title and data).
flmaxcol	smallint	The number of columns occupied by the field (title and data).
flposy	smallint	The y coordinate of the upper left corner of the field.
flposx	smallint	The x coordinate of the upper left corner of the field.
fldatawidth	smallint	The width of the data entry area for the field. If field format is c20.10, fldatawidth is 10.
fldatalin	smallint	The y coordinate position of the data entry area relative to the upper left corner of the field.
fldatacol	smallint	The x coordinate position of the data entry area relative to the upper left corner of the field.
fltitle	varchar(50)	The field title.

Column Name	Data Type	Description
fltitcol	smallint	The x coordinate position of the title relative to the upper left corner of the field.
fltitlin	smallint	The y coordinate position of the title relative to the upper left corner of the field.
flintrp	smallint	Reserved for future use.
fldflags	integer	The field attributes, such as box and reverse video. Valid (octal) values are:
		1 boxed field
		04 query-only
		10 keep previous values
		20 mandatory
		40 no row lines (table field)
		100 force lowercase
		200 force uppercase
		400 reverse video
		1000 blinking
		2000 underline
		4000 change intensity
		10000 no autotab
		20000 no echo
		40000 no column title (table field only)
fldflags (cont'd.)		200000 foreground color 1
		400000 foreground color 2
		1000000 foreground color 3
		2000000 foreground color 4
		4000000 foreground color 5
		10000000 foreground color 6
		20000000 foreground color 7
		100000000 invisible
		10000000000 row highlight (table

Column Name	Data Type	Description
		field)
fld2flags	integer	More attributes for the field, including scrolling: 0100 scrollable 01000 display-only 04000 derived field
fldfont	smallint	Reserved for future use.
fldptsz	smallint	Reserved for future use.
fldefault	varchar(50)	The default value for the field.
flformat	varchar(50)	The display format for the field (for example, c10 or f10.2).
flvalmsg	varchar(100)	The message to be displayed if the validation check fails.
flvalchk	varchar(240)	The validation check for the field.

fltype	smallint	Indicates if the record describes a regular field, a table field or a column in a table field. Possible values are:
		0 simple field;
		1 table field;
		2 table field column;

flsubseq	smallint	A unique identifying record number with respect to the set of records that describe all the fields in a form.

ii_forms Catalog

The ii_forms catalog contains one row for each form in a database.

The ii_forms catalog is structured as btree unique, keyed on the object_id column:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying this form in the ii_objects catalog. Other information about the form (such as name, owner, and object class) is stored in the ii_objects catalog.
frmaxcol	smallint	The number of columns the form occupies.

Column Name	**Data Type**	**Description**
frmaxlin	smallint	The number of lines the form occupies.
frposx	smallint	The x coordinate for the upper left corner of the form.
frposy	smallint	The y coordinate for the upper left corner of the form.
frfldno	smallint	For forms saved before release 6.3/01, contains the number of updatable regular and table fields in the form. For forms saved with or after release 6.3/01, contains the number of regular and table fields in the form.
frnsno	smallint	For forms saved before release 6.3/01, the number of display-only regular fields in the form.
frtrimno	smallint	The number of trim and box graphic trim strings in the form.
frversion	smallint	Version number of the form.
frscrtype	smallint	Reserved for future use.
frscrmaxx	smallint	Reserved for future use.
frscrmaxy	smallint	Reserved for future use.
frscrdpix	smallint	Reserved for future use.
frscrdpiy	smallint	Reserved for future use.
frflags	integer	The attributes of the form, such as whether this a pop-up or normal form. Valid (octal) values are: 1 Display form with single-line border 200 Display form as pop-up 4000 Display form in narrow -screen mode 10000 Display form in wide-screen mode
fr2flags	integer	More attributes for the form. Currently unused.
frtotflds	integer	The total number of records in the ii_fields catalog for the form.

ii_trim Catalog

The ii_trims catalog contains the trim strings and box graphic trim for a form. There is one row for each trim string and for each box graphic trim.

The ii_trim catalog is structured as compressed btree unique on the object_id, trim_col and trim_lin columns:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying the form this trim string belongs to in the ii_objects catalog. Other information about the form (such as name, owner, and object class) is stored in the ii_objects catalog.
trim_col	smallint	The x coordinate for the starting position of the trim string or box graphic trim.
trim_lin	smallint	The y coordinate for the starting position of the trim string or box graphic trim.
trim_trim	varchar (150)	The actual trim string or encoding of box graphic trim size (number of rows and columns).
trim_flags	integer	Attributes of the trim string: 01 box graphic trim 0400 reverse video 01000 blinking 02000 underline 04000 change intensity 0200000 foreground color 1 0400000 foreground color 2 01000000 foreground color 3 02000000 foreground color 4 04000000 foreground color 5 010000000 foreground color 6 020000000 foreground color 7
trim2_flags	integer	More attributes for the trim string. Currently unused.
trim_font	smallint	Reserved for future use.
trim_ptsz	smallint	Reserved for future use.

ABF System Catalogs

The extended system catalogs that are required by the ABF system are as follows:

- ii_abfclasses
- ii_abfdependencies
- ii_abfobjects
- ii_sequence_values

ii_abfclasses Catalog

The ii_abfclasses catalog contains information on the attributes of ABF record types. Name, owner, and class information is stored in ii_objects; information about the record types is stored in ii_abfobjects:

Column Name	Data Type	Description
appl_id	integer	Object ID of the ABF application that contains this object.
class_id	integer	Object ID of the record type containing the attributes.
catt_id	integer	Object ID in the ii_objects catalog.
class_order	smallint	Unused.
adf_type	integer	Integer code representing the type of the attribute (user frames, procedures). Valid values are listed below; NULLable data types are represented as the negative of the listed value: 0 none 3 date 5 money 30 integer 31 floating point 37 text 40 string
type_name	varchar(32)	Description of adf_type.
adf_length	integer	Length of return type.
adf_scale	integer	Scale factor of return type.

ii_abfdependencies Catalog

The ii_abfdependencies table describes how the objects listed in the ii_abfobjects table depend on each other, and on other database objects such as reports. To get a list of application dependencies, you must join this table with ii_objects over the object_id column. For an example of joining ii_abfdependencies with ii_objects, see Sample Queries for the Extended System Catalogs for SQL (see page 495).

The ii_abfdependencies catalog is structured as btree, keyed on the abfdef_applid and object_id columns:

Column Name	Data Type	Description
object_id	integer	Object ID of an ABF application object that is dependent on another object. Other information about this object (such as name, owner and object class) is stored in the ii_objects and ii_abfobjects catalogs.
abfdef_applid	integer	Object ID of the ABF application that contains this object.
abfdef_name	varchar(32)	Name of depended-upon object. If the row indicates a frame or procedure's dependence on an ii_encodings entry, the name is fid plus the object ID of the ii_encodings entry. If the row only exists to avoid an outer join problem between this table ii_abfobjects and ii_objects, the name is DUMMY.
abfdef_owner	varchar(32)	Catalog updater. Present for naming consistency across user interface catalogs, not currently important for correct ABF operation.
object_class	integer	Object manager class of depended upon object.
abfdef_deptype	integer	Type of dependency: 3502 Dependent on a database object. 3503 Call with no use of return code. 3504 Call with return code. 3505 Menu dependency.

Column Name	Data Type	Description
		3506 Dependency on a global variable. 3507 Dependency on a record type. 3508 Dependency on table form [type of table] declaration. 3509 Dependency on form [type of form] declaration. 3510 Dependency on form [type of table field] declaration.
abf_linkname	varchar(32)	Vision Menu item text for menu dependency.
abf_deporder	integer	Vision Order of menu dependency.

ii_abfobjects Catalog

The ii_abfobjects catalog contains ABF-specific information on ABF objects. Name, owner, and class information on each object is contained in the ii_objects catalog, and is obtained by joining ii_abfobjects with ii_objects over the ID column. For an example of joining ii_abfobjects with ii_objects, see the Querying the Extended System Catalogs for SQL (see page 495). The ABF application is also considered an object, and corresponds to a row in which applid=object_id.

The ii_abfobjects catalog is structured as compressed btree unique on the applid and object_id columns:

Column Name	Data Type	Description
applid	integer	Object ID of the ABF application that contains this object.
object_id	integer	Unique identifier (object ID) for identifying this object in the ii_objects catalog. Other information about this object (such as name, owner, and object class) is stored in the ii_objects catalog.
abf_source	varchar (180)	Source file name (without path) for objects with source files; source path for the application.
abf_symbol	varchar(32)	Linker symbol corresponding to object, for objects that are compiled

Column Name	Data Type	Description
		and linked (compiled forms, user frames, procedures).
retadf_type	smallint	Integer code (ADT type) for return type of objects that have return types (user frames, procedures). Possible values are listed below with NULLable data types being the negative of the listed value: 0 none 3 date 5 money 30 integer 31 floating point 37 text 40 string
rettype	varchar(32)	A textual description of retadf_type.
retlength	integer	Length of return type.
retscale	integer	Scale factor of return type.
abf_version	smallint	Release number for latest update of object. Contains 0 for release 5.
abf_flags	integer	32-bit flag variable; the flags describe the state of the component.
abf_arg1	varchar(48)	Object specific (field 1): applications: Contains the executable name. report or QBF frames: Command line flags (if specified). procedures: The host language (descriptive string only, derived from fill extension). constants: The language of the constant (for example, English or French).
abf_arg2	varchar(48)	Object specific (field 2). If object is: An application: this field contains its default starting frame. Report-writer: this field contains the output destination (file) QBF frames: this field contains the joindef/table flag.

Column Name	Data Type	Description
abf_arg3	varchar(48)	Object-specific field 3. For applications: The link option file.
abf_arg4	varchar(48)	Object-specific field 4. For applications: specifies the query language (QUEL or SQL). For 3GL or 4GL frame: contains the date of the last unsuccessful compile.
abf_arg5	varchar(48)	Object-specific field 5. For applications: Contains the role under which the application runs. For Vision frames: Contains the date and time the form was generated.
abf_arg6	varchar(48)	Object-specific field 6. For Vision frames: Contains the date and time the code was generated.

ii_sequence_values Catalog

The ii_sequence_values catalog is used by the 4GL sequence_value function to generate a series of increasing values (for example, in an application that automatically assigns the next invoice number).

The format of ii_sequence_values is as follows:

Column Name	Data Type	Description
sequence_owner	varchar(32)	The owner of the table that receives the sequence value.
sequence_table	varchar(32)	The table that receives the sequence value.
sequence_column	varchar(32)	The column that receives the sequence value.
sequence_value	integer	The last value generated by the sequence_value function.

QBF System Catalogs

The QBF system requires the following extended system catalogs:

- ii_joindefs
- ii_qbfnames

ii_joindefs Catalog

The ii_joindefs catalog contains additional information about join definitions (JoinDefs) used in QBF. Basic information about the JoinDef is contained in a row in the ii_objects catalog. Each JoinDef can have several rows in ii_joindefs associated with it. There are four type of records in ii_joindefs, identified by the qtype column. The ii_joindefs catalog is structured as compressed btree unique on the object_id and qtype columns:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying this JoinDef in the ii_objects catalog. Other information about the JoinDef (such as its name, owner, and object class) is stored in the ii_objects catalog.
qtype	integer	The low order byte of this column indicates the record type of this row, as follows: 0—Indicates if a table field is used in the JoinDef. 1—Table information. 2—Column information. 3—Join information. The high order byte is used as a sequence number for multiple entries of a particular record type. Each JoinDef has exactly one row with qtype = 0; it has one row with qtype = 1 for each table used in the JoinDef; it has one row with qtype = 2 for each field displayed in the JoinDef; it has one row with qtype = 3 for each pair of columns joined in the JoinDef.
qinfo1	varchar(32)	If qtype = 0, qinfo1 indicates if the

Column Name	Data Type	Description
		JoinDef is built with a table field format (Y = yes, N = no). If qtype = 1, qinfo1 contains the name of a table used in the JoinDef. If qtype = 2, qinfo1 contains a correlation name (range variable) for the table used in the JoinDef that contains the column named in qinfo2. If qtype = 3, qinfo1 contains a correlation name (range variable) for a column named in qinfo2 that is joined to the column referenced in qinfo3 and qinfo4.
qinfo2	varchar(32)	If qtype = 0, qinfo2 is not used. If qtype = 1, qinfo2 indicates whether the table named in qinfo1 is a Master (0) or Detail (1) table. If qtype = 2, qinfo2 contains the name of the column to be used in conjunction with the correlation name in qinfo1. If qtype = 3, qinfo2 contains the name of the column to be joined to the column referenced in qinfo3 and qinfo4.
qinfo3	varchar(32)	If qtype = 0, qinfo3 is not used. If qtype = 1, qinfo3 contains a correlation name (range variable) for the table named in qinfo1. If qtype = 2, qinfo3 contains the field name in the form corresponding to the column identified by qinfo2. If qtype = 3, qinfo3 contains a correlation name (range variable) for a column named in qinfo4 that is joined to the column referenced in qinfo1 and qinfo2.
qinfo4	varchar(32)	If qtype = 0, qinfo4 is not used. If qtype = 1, qinfo4 contains the delete rules for the table named in qinfo1 (0 = no, 1 = yes). If qtype = 2, qinfo4 contains the status codes for the column identified by qinfo1 and qinfo2. These status codes are expressed as a 3-character text string; the first character denotes update rules for values in this column (0 = no, 1 = yes); the second character denotes whether this column is part of a join (0 = no, 1 = yes); the third character denotes whether this column is a displayed column (0 = no, 1 = yes). Typically, if the column is not part of a

Column Name	Data Type	Description
		join the third character is not used by QBF. If qtype = 3, qinfo4 contains the name of the column to be joined to the column referenced in qinfo1 and qinfo2.
qinfo5	varchar(32)	The owner of the table referenced by the joindef.

ii_qbfnames Catalog

The ii_qbfnames catalog contains information used by QBF on the mapping between a form and a corresponding table or JoinDef.

The ii_qbfnames catalog is structured as compressed btree unique on the object_id column:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying this QBFName in the ii_objects catalog. Other information about this QBFName (such as its name, owner, and object class) is stored in the ii_objects catalog.
relname	varchar(32)	The name of a table or JoinDef.
relowner	varchar(32)	the owner of the table referenced in the QBFName.
frname	varchar(32)	The name of a form corresponding to the table or JoinDef.
qbftype	smallint	Indicates if the QBFName is mapping a form to a table (value 0) or JoinDef (value 1).

Report-Writer System Catalogs

The Report-Writer requires the following extended system catalogs:

- ii_rcommands
- ii_reports

ii_rcommands Catalog

The ii_rcommands catalog contains the formatting, sorting, break, and query commands for each report, broken down into individual commands.

The ii_rcommands catalog is structured as compressed btree unique, keyed on the object_id, rcotype, and rcosequence columns:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying the report this command belongs to in the ii_objects catalog. Other information about the report (such as name, owner and object class) is stored in the ii_objects catalog.
rcotype	char(2)	Report command type. Valid values are: TA—Table for a .data command. SQ—Piece of SQL query for the .query command. QU—Piece of QUEL query for the query command. SO—Sort column for a .sort command. AC—Report formatting or action command. OU—Output file name, if specified. BR—break command information. DE—declare statement information.
rcosequence	smallint	The sequence number for this row, in the rcotype.
rcosection	varchar(12)	The section of the report, such as header or footer, to which the commands refer if rcotype is AC. If rcotype is QU or SQ, this refers to the part of the query described. For other values of rcotype, this field is unused.
rcoattid	varchar(32)	If rcotype is AC, this indicates either the column name associated with the footer/header section or contains the value PAGE or REPORT or DETAIL. If SO, this is the name of the sort

Column Name	Data Type	Description
		column. If QU, the range variable names are put in this column. If BR, the name of the break column is put in this column. If DE, the name of the declared variable is put in this column.
rcocommand	varchar(12)	Primarily used for the names of the formatting commands when rcotype is AC. Also used by SO rcotype to indicate that the sort column is also a break column.
rcotext	varchar(100)	If the rcotype is AC, this contains the text of the formatting command. If type OU, this contains the name of the output file. If QU or SQ, this contains query text. If TA, this contains the table name. If SO, this contains the sort order. If DE, this contains the text of the declaration. If BR, this is unused.

ii_reports Catalog

The ii_reports catalog contains information about reports. There is one row for every report in the database. Both reports created through RBF and reports created through sreport contain entries in ii_reports. For an example of a query that selects information about reports, see Sample Queries for the Extended System Catalogs for SQL (see page 495).

The ii_reports table is structured as btree unique on the object_id column:

Column Name	Data Type	Description
object_id	integer	Unique identifier (object ID) for identifying this report in the ii_objects catalog. Other information about the report (such as name, owner, and object class) is stored in the ii_objects catalog.
reptype	char(1)	The method used to create the report; S if the report was created by sreport, and F if the report was

Column Name	Data Type	Description
		created by RBF.
replevel	integer	The release level of the report, shown on the copyrep header (not present in earlier releases). Used internally by Report tools. 0 for earlier releases 1 for the current release The default is 0.
repacount	smallint	The number of rows in the ii_rcommands catalog with an rcotype of AC. This is used for internal consistency.
repscount	smallint	The number of rows in the ii_rcommands catalog with an rcotype of SO. This is used for internal consistency.
repqcount	smallint	The number of rows in the ii_rcommands catalog with an rcotype of QU. This is used for internal consistency.
repbcount	smallint	The number of rows in the ii_rcommands catalog with an rcotype of BR. This is used for internal consistency.

Vision System Catalogs

Vision requires the following catalogs. These catalogs comprise the APPLICATION_DEVELOPMENT_3 module.

- ii_framevars
- ii_menuargs
- ii_vqjoins
- ii_vqtabcols
- ii_vqtables

ii_framevars Catalog

The ii_framevars catalog describes the local variables and hidden table fields of a frame:

Column Name	Data Type	Description
object_id	integer	Object ID of the frame.
fr_seq	smallint	Sequence number (for ordering).
var_field	varchar(32)	Field name.
var_column	varchar(32)	Column name (if field is a table field).
var_datatype	varchar(105)	Data type of the field.
var_comment	varchar(60)	Comment for the variable.

ii_menuargs Catalog

The ii_menuargs catalog describes the arguments to be passed to a called frame for a given menu choice:

Column Name	Data Type	Description
object_id	integer	Object ID of frame.
mu_text	varchar(32)	Menu item text.
mu_seq	smallint	Sequence number beginning at 0 (for argument ordering).
mu_field	varchar(32)	Field name in called frame to assign value to. If field name is of the form X.Y, this contains the X portion only.
mu_column	varchar(32)	Portion of field name in called frame to assign value to. Only used when field name is of form X.Y, in which case this contains the Y portion.
mu_expr	varchar(240)	4GL expression (field, constant, byref(), etc.) in the parent frame to assign to field in the called frame.

ii_vqjoins Catalog

The ii_vqjoins catalog describes the joins involved in a visual query:

Column Name	Data Type	Description
object_id	integer	Object ID of the frame.
vq_seq	smallint	Sequence number (for ordering).
join_type	smallint	Type of join specified by this row. 0 = Master/Detail join. 1 = Master/Lookup join. 2 = Detail/Lookup join.
join_tab1	smallint	Index to table 1 of the join. Relative table number in visual query beginning with 0.
join_tab2	smallint	Index to table 2 of the join. Relative table number in visual query beginning with 0. (table 2 is always below table 1 in visual query).
join_col1	smallint	Join column for table 1. Index into array of columns for table 1. (Same as ii_vqtabcols.vq_seq).
join_col2	smallint	Join column for table 2. Index into array of columns for table 2. (Same as ii_vqtabcols.vq_seq).

ii_vqtabcols Catalog

The ii_vqtabcols catalog describes the columns of the tables involved in a visual query:

Column Name	Data Type	Description
object_id	integer	Object ID of the frame.
vq_seq	smallint	Column sequence number (for ordering).
tvq_seq	smallint	Sequence number of table in visual query from ii_vqtables.
col_name	varchar(32)	Column name.
ref_name	varchar(32)	Name used on the form for field containing data.

Column Name	Data Type	Description
adf_type	smallint	Type information for column. See description of iicolumns. column_ingdatatype for details.
adf_length	integer	Column size in bytes.
adf_scale	integer	Currently not used.
col_flags	integer	This column contains multiple pieces of information about the column in a bitmap format. The following values are present (expressed in Hex): 1 = Column is to be used on form/report. 2 = Column is joined to a detail table and must be displayed. 4 = Column is joined to a lookup table and must be displayed. 8 = Column is a subordinate join field; therefore it cannot be displayed. 0x10 = Column is sequenced (generate new surrogate key value for INSERT statements). 0x20 = Column is descending (for sort). 0x40 = Column is part of the table's unique key. 0x100 = Set if column allows defaults.
col_sortorder	smallint	Sort order for this column. Set to 0 if not part of sort sequence. For lookup tables, this gives the order of the column in the lookup screen.
col_info	varchar(240)	Information entered by developer for this column in visual query. For Browse & Update frames, this is a query restriction and is added to the WHERE clause of the SELECT statement. For Append frames, this gives default value information and is either used in 4gl assignment statements for a displayed column, or in the INSERT statement for a not-displayed column.

ii_vqtables Catalog

The ii_vqtables catalog describes the tables involved in a visual query:

Column Name	Data Type	Description
object_id	integer	Object ID of the frame.
vq_seq	smallint	Order of table in visual query.
vq_mode	smallint	This column contains multiple pieces of information about the frame as a whole, in a bitmap format (although note that the first 4 entries below are mutually exclusive; only one of them can appear). Can contain the following values (in Hex): 0 = Frame has no tables (menu frame). 1 = Master/Detail frame. 2 = Master only in a table field. 3 = Master only in simple fields. 0x10 = If set, the Qualification Processing frame behavior is Disabled (can only be set for Browse & Update frames) 0x20 = If set, the Next Master Menuitem frame behavior is Disabled (can only be set for Browse & Update frames) 0x40 = If set, the"Hold Locks on Displayed Data frame behavior is set to Yes (can only be set for Update frames).
tab_name	varchar(32)	Table name.
tab_owner	varchar(32)	Table owner.
tab_section	smallint	Visual query section table is in. 0 = table is in master section. 1 = table is in detail section.
tab_usage	smallint	How this table is used in the visual query. 0 = Append table. 1 = Update table. 2 = Browse table. 3 = Lookup table.
tab_flags	integer	Bitmap flag that indicates the frame

Column Name	Data Type	Description
		behaviors in the visual query. Valid values (in Hex): 0x1 = For lookup table: lookup requires a qualification screen. For update table: insertions are allowed into the table field (only relevant to the detail table, and to masters in table field) 0x2 = OK to Delete data in this table. 0x4 = If set: update of join field cascades to detail; if clear: update of join field not allowed if details exist. 0x8 = If set: delete of master cascades to detail; if clear: delete of master not allowed if details exist. 0x10 = Table does not have a unique key. 0x20 = DBMS handles referential integrity on details when join field is changed. Generated code updates master table only. 0x40 = DBMS handles referential integrity on details when master is deleted.

Additional Vision Catalog Information

The Application Flow Diagram and Escape Code provide additional Vision Catalog information.

Vision's Application Flow Diagram is built from menu item information in the ii_abfdependencies catalog. See ii_abfdependencies Catalog (see page 505).

Frame Escape Code is stored in the ii_encodings catalog. See ii_encodings Catalog (see page 490).

All escape code for a frame is combined into one (possibly resequenced) entry in iiencodings. Each piece of escape code in the entry is preceded by a type code. For example:

```
 1 = form-start
 2 = form-end
 3 = query-start
 4 = query-new-date
 5 = query-end
 6 = append-start
 7 = append-end
 8 = update-start
 9 = update-end
10 = delete-start
11 = delete-end
12 = menu-start
13 = menu-end
14 = field-entry
15 = field-change
16 = field-exit
17 = user-menuitem
```

DBMS System Catalogs

The table names of the DBMS System Catalogs can be used as arguments to the sysmod command. These tables are not supported for any other use.

System Catalogs for All Databases

Following are the table names of DBMS System Catalogs for all databases. The information in these catalogs is accessed by selecting from the standard catalogs:

Catalog	Description
iiattribute	Describes the properties of each column of a table.
iidbdepends	Describes the dependencies between views or

Catalog	Description
	protections and their base tables.
iidbms_comment	Contains text for comments on tables or columns.
iidefault	Stores default values used by any attribute (column) in any table residing in this database.
iidevices	Describes additional locations when a user table spans more than one location.
iidistcol	Lists the partitioning columns for partitioned tables.
iidistscheme	Contains information about partitioning schemes for partitioned tables.
iidistval	Contains the partitioning values and directives for LIST or RANGE partitioned tables.
iievent	Contains database event information.
iiextended_relation	Contains information about the association between base tables and the extended tables used to store long data types.
iigw06_attribute	Security audit gateway catalogs.
iigw06_relation	Security audit gateway catalogs.
iihistogram	Contains database histograms that are collected by the optimizedb program.
iiindex	Describes all the indexes for a table.
iiintegrity	Contains information about the integrities applied to tables.
iikey	Contains information about key attributes for unique and referential constraints.
iipartname	Contains logical partition names for partitioned tables.
iipriv	Contains information about privileges and their dependent objects.
iiprivlist	Contains list of privilege names.
iiprocedure	Contains information about database procedures.
iiprocedure_ parameter	Contains information about database procedure parameters.
iiprotect	Contains information about the protections applied to tables.
iiqrytext	Contains the actual query text for views, protections, and integrities.

Catalog	Description
iirel_idx	An index table that indexes the iirelation table by table name and owner.
iirelation	Describes each table in the database.
iirule	Contains information about rules in the database.
iischema	Contains the schema name, owner and ID.
iisecalarm	Contains information about security alarms.
iisectype	A lookup table for security event types.
iisequence	Contains information about all sequences defined in the database.
iistatistics	Contains database statistics that are collected by the optimizedb program.
iisynonym	Contains information on the synonyms that have been defined for tables, views and indexes.
iitree	Contains the DBMS internal representation of query text for views, protections, and integrities.
iixdbdepends	An index table used to find the rows that reference a dependent object in the iidbdepends catalog.

System Catalogs for iidbdb

Following are the table names of DBMS System Catalogs that exist only in the master database (iidbdb). These tables can be used as arguments to the sysmod command. They are not supported for any other use.

The information in these catalogs is accessed by selecting from standard catalogs when connected to the iidbdb database:

Catalog	Description
iicdbid_idx	Index on iistar_cdbs
iidatabase	Various attributes of the databases existing in this installation.
iidbid_idx	Index on iidatabase.
iidbpriv	Contains information about database privileges.
iidbdb_netcost	Contains costing information used by Ingres Star.
iidbdb_nodecost	Contains costing information used by Ingres Star.

Catalog	Description
iiextend	Information about what locations databases have been extended to.
iigw07_attribute	Ingres Management Architecture (IMA) catalog.
iigw07_index	Ingres Management Architecture (IMA) catalog.
iigw07_relation	Ingres Management Architecture (IMA) catalog.
iilocations	Storage locations defined in this installation.
iiprofile	User profiles defined in this installation.
iirole	Roles defined in this installation
iirolegrant	Information about which grantees have role privileges.
iisecuritystate	Information relating to the security state of this installation.
iistar_cdbs	Information about Ingres Star coordinator databases in this installation.
iiuser	Users defined in this installation
iiusergroup	Group definitions for this installation

Miscellaneous System Catalogs

Following are the table names of system catalogs that are created by default and are owned by $ingres, but do not fit into any of the previous categories:

Catalog	Description
iiocolumns	An old system catalog interface that has been replaced by iicolumns.
iiotables	An old system catalog interface that has been replaced by iitables.
iistar_cdbinfo	A standard catalog interface to data that describes coordinator databases for Ingres Star. For more information, see the *Ingres Star User Guide*.

Appendix B: Ingres Limits

This section contains the following topics:

Summary of Limits

Following is a summary of Ingres limits:

Parameter	Limit
Maximum database size	1,000 EB (exabytes)
Maximum tables in a database	65,536
Maximum files per database	2 billion
Maximum files per instance	Limit based on file systems.
Maximum page size	64 KB
Maximum cache buffers	4 GB on 32-bit systems 16 EB on 64-bit systems Limited by hardware/OS memory architecture.
Maximum rows per table	34.5 trillion
Maximum row size per table	256 KB Limit does not include LOBs.
Maximum fields (columns) per table	1024
Maximum indexes per table	126
Maximum file size per index	512 GB
Maximum fields (columns) per index	32
Maximum integer size	64 bits
Maximum decimal precision	39 digits
Maximum float precision	Hardware dependent
Maximum length of a character field	32,000 bytes If II_CHARSETxx=UTF8,16,000 bytes
Maximum length of a varchar field	32,000 bytes

Parameter	Limit
	If II_CHARSETxx=UTF8,16,000 bytes
Maximum size of a CLOB	2 GB
Maximum size of a BLOB	2 GB
Maximum size of an SQL statement	Unlimited
Maximum members in an IN list	Unlimited
Maximum logical operators in a WHERE clause	Unlimited
Maximum join conditions	Unlimited
Maximum tables in a join statement	126
Unicode support	UTF-8 and UCS-2 (UTF-16 without surrogate support)
XML support	Consume and publish

Index