Performance Guide

for Informix Extended Parallel Server

Version 8.3
December 1999
Part No. 000-6543

Published by Informix® Press

Informix Corporation
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Documentation Team: Diana Chase, Mary Kraemer, Hanna Metzger, Virginia Panlasigui

Table of Contents

Introduction

Chapter 1 Performance Basics

Chapter 2 Performance Monitoring

Chapter 3 Effect of Configuration on CPU Use

Chapter 4 Effect of Configuration on Memory Use

Chapter 8 Locking

Chapter 9 Fragmentation Guidelines

Introduction

In This Introduction

This Introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual provides information about how to configure and operate Informix Extended Parallel Server to improve overall system throughput and how to improve the performance of SQL queries.

Types of Users

This manual is for the following users:

- Database administrators
- Database server administrators
- Database application programmers
- Performance engineers

This manual assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming
- Some experience with database server administration, operating-system administration, or network administration

If you have limited experience with relational databases, SQL, or your operating system, refer to your *Getting Started* manual for a list of supplementary titles.

Software Dependencies

This manual assumes that you are using Informix Extended Parallel Server, Version 8.3, as your database server.

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a Global Language Support (GLS) locale.

The examples in this manual are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Informix Guide to GLS Functionality*.

Demonstration Databases

The DB-Access utility, which is provided with your Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in Informix manuals are based on the **stores_demo** database.
- The **sales_demo** database illustrates a dimensional schema for data warehousing applications. For conceptual information about dimensional data modeling, see the *Informix Guide to Database Design and Implementation*.

For information about how to create and populate the demonstration databases, see the *DB-Access User's Manual*. For descriptions of the databases and their contents, see the *Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory.

New Features

For a comprehensive list of new database server features, see the release notes. This section lists new features relevant to this manual.

The Version 8.3 features that this manual describes fall into the following areas:

- Data-migration enhancements
- Configuration enhancements
- Table-fragmentation enhancements
- Performance enhancements
- New SQL functionality
- Utility features
- Version 8.3 features from Informix Dynamic Server, Version 7.30

Data-Migration Enhancements

This manual describes the following data-migration enhancements to Version 8.3 of Extended Parallel Server:

- Version 8.3 to Version 8.21 reversion
- Data movement from Version 7.2 or Version 7.3 database servers to Version 8.3

Configuration Enhancements

This manual describes the following configuration enhancements to Version 8.3 of Extended Parallel Server:

- Increased maximum number of chunks, dbspaces, and dbslices
- Configurable page size
- 64-bit very large memory (VLM)
- Increased maximum chunk size

Table-Fragmentation Enhancements

This manual describes the following table-fragmentation enhancements to Version 8.3 of Extended Parallel Server:

- ALTER FRAGMENT attach with remainders
- ALTER TABLE to add, drop, or modify a column
- Range fragmentation

Performance Enhancements

This manual describes the following performance enhancements to Version 8.3 of Extended Parallel Server:

- Coarse-grain index locks
- Fuzzy checkpoints

New SQL Functionality

This manual describes the following new SQL functionality in Version 8.3 of Extended Parallel Server:

- CASE statement in Stored Procedure Language (SPL)
- DELETE ... USING statement to delete rows based on a table join
- Globally detached indexes
- Load and unload simple large objects to external tables
- MIDDLE function
- Referential integrity for globally detached indexes
- TRUNCATE statement

Utility Features

This manual describes the following new options for the **onutil** utility in Version 8.3 of Extended Parallel Server.

- **onutil** CHECK, without locks
- **onutil** CHECK, repair improvements
- **onutil** ALTER DBSLICE
- **onutil** ALTER LOGSLICE

Version 8.3 Features from Version 7.30

This manual describes the following features from Version 7.30 of Dynamic Server in Version 8.3 of Extended Parallel Server:

- Ability to retain update locks
- ALTER TABLE to add or drop a foreign-key constraint
- Constraints on columns other than the fragmentation column
- Slow ALTER TABLE
- Triggers
- Memory-resident tables

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Sample-code conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** `italics`	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` `monospace`	Information that the product displays and information that you enter appear in a monospace typeface.

(1 of 2)

Convention	Meaning
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

(2 of 2)

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons.

Comment icons identify three types of information, as the following table describes. This information always appears in italics.

Icon	Label	Description
	***Warning:***	Identifies paragraphs that contain vital instructions, cautions, or critical information
	***Important:***	Identifies paragraphs that contain significant information about the feature or operation that is being described
	***Tip:***	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature.

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using DB-Access, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement.

Tip: *Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.*

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message documentation
- Documentation notes, release notes, and machine notes
- Related reading

On-Line Manuals

An Answers OnLine CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine.

Informix on-line manuals are also available on the following Web site:

```
www.informix.com/answers
```

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com. Please provide the following information when you place your order:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Documentation

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions.

To read error messages and corrective actions, use one of the following utilities.

Utility	Description
finderr	Displays error messages on line
rofferr	Formats error messages for printing

Instructions for using the preceding utilities are available in Answers OnLine. Answers OnLine also provides a listing of error messages and corrective actions in HTML format.

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following sections describe the on-line files that supplement the information in this manual. Please examine these files before you begin using your database server. They contain vital information about application and performance issues.

The following on-line files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

On-Line File	Purpose
PERFDOC_8.3	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
SERVERS_8.3	The release notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
XPS_*x.y*	The machine notes file describes any special actions that you must take to configure and use Informix products on your computer. Machine notes are named for the product described.

Related Reading

The following publications provide additional information about the topics that this manual discusses. For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started* manual.

- *Measurement and Tuning of Computer Systems* by Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner (Prentice-Hall, Inc., 1983)
- *High Performance Computing* by Kevin Dowd (O'Reilly & Associates, Inc., 1993)

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Let us know what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send electronic mail, our address is:

doc@informix.com

The **doc** alias is reserved exclusively for reporting errors and omissions in our documentation.

We appreciate your suggestions.

Chapter

1

Performance Basics

In This Chapter

This chapter provides an overview of performance measurement and tuning for Informix Extended Parallel Server. Performance measurement and tuning encompass a broad area of research and practice.

This manual discusses only performance tuning issues and methods that are relevant to daily database server administration and query execution. For an introduction to basic database design, refer to the *Informix Guide to Database Design and Implementation*. For a general introduction to performance tuning, refer to the texts listed in "Related Reading" on page 13.

Information in this manual can help you perform the following tasks:

- Monitor system resources that are critical to performance
- Identify database activities that affect these critical resources
- Identify and monitor queries that are critical to performance
- Use the database server utilities for performance monitoring and tuning
- Eliminate performance bottlenecks in the following ways:
  - Balancing the load on system resources
  - Adjusting the configuration of your database server
  - Adjusting the arrangement of your data
  - Allocating resources for decision-support queries
  - Creating indexes to speed up retrieval of your data

This chapter provides information about the following topics:

- Parallel processing
- Decision support applications
- Performance goals

- Performance measurements and resource utilization
- Maintenance of good performance
- Topics that are not covered in this manual

Parallel Processing

One of the most important factors in performance tuning for Extended Parallel Server is balancing resource use and parallel processing within and across coservers.

This brief description of the parallel-processing architecture and performance advantages of Extended Parallel Server provides helpful background information for understanding parallel processing features.

Parallel-Processing Architecture

The parallel-processing architecture of Extended Parallel Server provides high performance for database operations on computing platforms that range from a single computer to parallel-processing platforms composed of dozens of computers. A parallel-processing platform is a set of independent computers that operate in parallel and communicate over a high-speed network, bus, or interconnect.

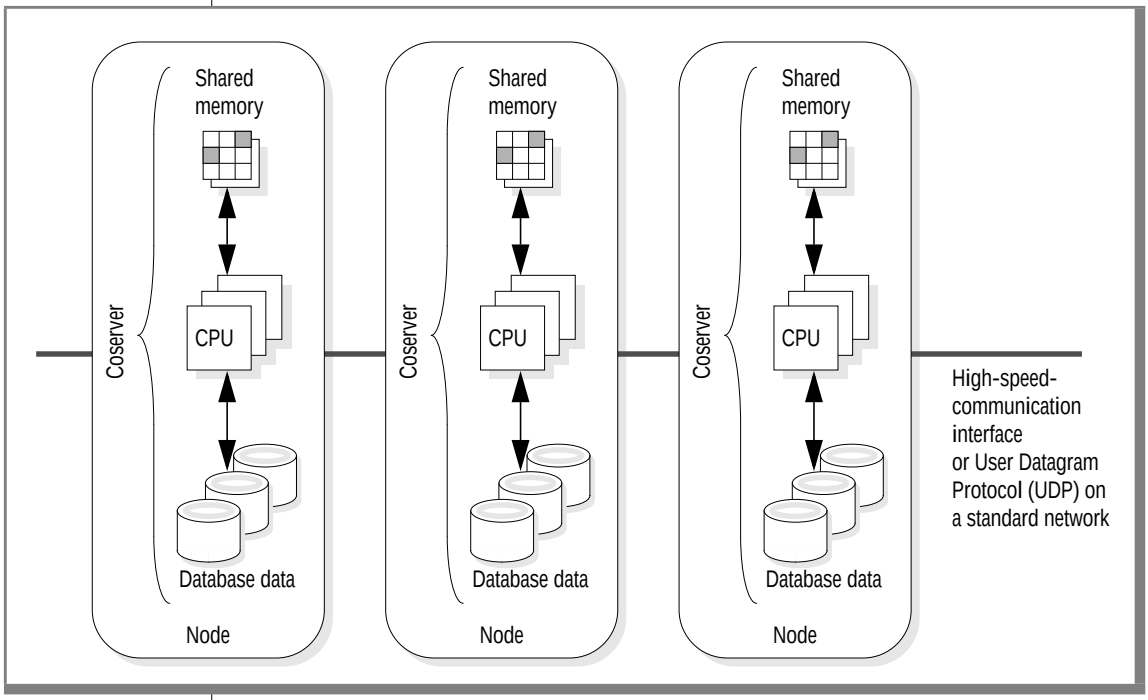
Each computer within a parallel-processing platform is referred to as a *node*. A node can be a uniprocessor or a symmetric multiprocessing (SMP) computer. Each computer manages its own disks, memory, and processors.

You configure Extended Parallel Server on a single computer or a parallel-processing platform as a set of one or more coservers. A *coserver* is the functional equivalent of a database server that operates on a single node. Each coserver performs database operations in parallel with the other coservers that make up a database server.

Each coserver independently manages its own resources and activities such as logging, recovery, locking, and buffers. This independent management of resources by each coserver is referred to as a *shared-nothing architecture*.

Figure 1-1 on page 1-5 illustrates the parallel-processing architecture of Extended Parallel Server. For information about the database server architecture, refer to your *Administrator's Guide*.

Figure 1-1
Database Server in a Shared-Nothing Environment



Performance Advantages

Extended Parallel Server provides the following performance advantages for decision-support system (DSS) and data-warehouse applications that access very large databases (VLDBs):

- Enhanced parallel execution
- Scalability
- Enhanced parallel access to VLDBs

Enhanced Parallel Access

You can partition a VLDB across multiple coservers with another feature, table fragmentation. Extended Parallel Server delivers maximum performance benefits when the data being queried is contained in fragmented tables that are distributed across disks that belong to many different coservers.

For a description of fragmented tables and how to use fragmentation for maximum performance, refer to Chapter 9, "Fragmentation Guidelines."

Enhanced Parallel Execution

Parallel execution is extremely useful for decision support queries, in which large volumes of data are scanned, joined, and sorted across multiple coservers.

A *connection coserver* is the coserver that manages a client connection to the database server. When a client database request requires access to data that resides in table fragments on other coservers, the other coservers are called *participating coservers*.

When the connection coserver determines that a query requires access to data that is fragmented across coservers, the database server divides the query plan into subplans for each of the participating coservers. This division is based on the fragmentation scheme of the tables and the availability of resources on the connection coserver and the participating coservers.

The database server distributes each query subplan to the pertinent coservers and executes the subplans in parallel. Each subplan is processed simultaneously with the others. Because each subplan represents a smaller amount of work than the original query plan, parallel execution across multiple coservers can drastically reduce the time that is required to process the query.

For example, consider the following SQL request:

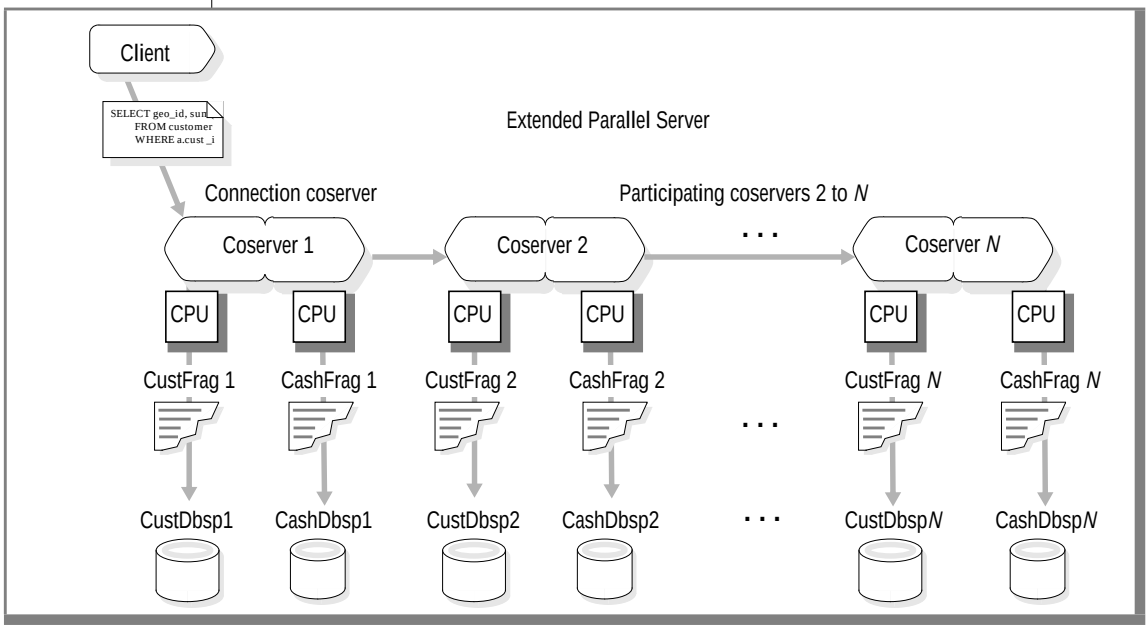
```
SELECT geo_id, sum(dollars)
   FROM customer a, cash b
   WHERE a.cust_id=b.cust_id
   GROUP BY geo_id
   ORDER BY SUM(dollars)
```

In this example, the connection and participating coservers perform the following tasks:

1. Each coserver scans relevant fragments of the **customer** table and the **cash** table in parallel.
2. Each coserver joins local rows from both the **customer** table and **cash** table by customer ID. Joins between rows that reside on different coservers can be performed by any of the participating coservers.
3. As coservers become free, they each perform some of the steps involved in selecting the geographic areas and dollar amounts that belong to particular customers and performing the group-by and order-by operations needed to complete the query. Each coserver sends its results to the connection coserver.
4. When the query is complete, the connection coserver returns the results to the client.

Figure 1-2 shows a client that is accessing a very large database that is fragmented across many coservers. **Coserver 1** is the connection coserver. **Coservers 1** through ***N*** are all participating coservers.

Figure 1-2
Client Query That Coservers Service



Scalability

Extended Parallel Server takes advantage of the underlying hardware parallelism to execute SQL operations and utilities in parallel. This ability to execute tasks in parallel provides a high degree of scalability for growing workloads.

Scalability has two aspects:

- Speed-up

  Speed-up is the ability to add computing hardware to achieve correspondingly faster performance for a DSS query or OLTP operation of a given volume.

  The ability to execute tasks in parallel promotes speed-up. For example, the database server on a parallel-processing platform with 10 CPUs can execute a complex query in approximately one-tenth the time of a single CPU system.

- Scale-up

  Scale-up is the ability to process a larger workload with a correspondingly larger amount of computing resources in a similar amount of time.

  If you fragment the data across multiple coservers, the database server uses the disk space, memory, and CPUs of these multiple coservers to process each fragment of data in parallel. The processing gains are directly proportional to the number of CPUs. For example, the database server on a parallel-processing platform with 10 CPUs can execute a complex query and access approximately 10 times the amount of data in the same time as a single CPU system.

  As your workload grows, you can add coservers to complete your DSS queries on the larger amount of data in about the same amount of time.

Extended Parallel Server provides highly linear scalability across a wide variety of computing platforms and workloads.

Decision Support

Two types of applications access data that is stored in a relational database:

- Decision-support system (DSS) applications and ad hoc queries
- On-line transaction processing (OLTP) applications

Figure 1-3 illustrates the difference between OLTP and DSS applications.

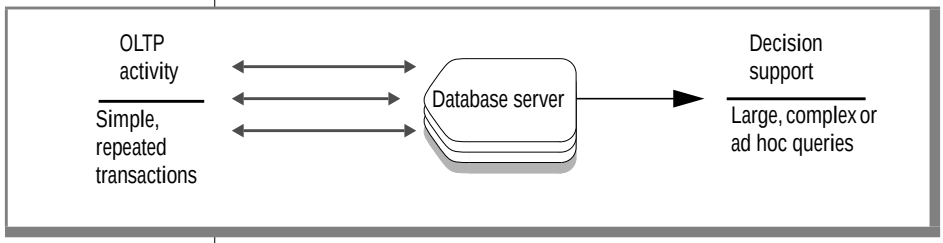


Figure 1-3
OLTP Versus DSS Operations

The built-in parallel processing capabilities and extended fragmentation schemes make Extended Parallel Server efficient for decision-support applications of all kinds.

Decision-Support Applications

Decision-support applications provide information that is used for strategic planning, decision making, and report preparation. These applications are frequently executed either at regular intervals in a batch environment or intermittently as *ad hoc* queries. Typical applications include payroll, inventory reporting, and end-of-period accounting reports.

Users can initiate ad hoc queries directly to get information such as:

- how many of the salesmen in the Western Region have already met their sales quota for this year, and who they are.
- how often telephone customers with billing addresses in the postal code beginning with 95 have used long-distance services to area codes 714, 805, 562, or 408.

You can run DSS queries directly against real-time OLTP data (also referred to as operational data) through gateways or middleware or by extracting, downloading, and reorganizing operational data to run large, memory-intensive queries against it. You can also retrieve decision-support information by downloading operational data to a personal computer for use with spreadsheets and end-user data-analysis tools.

Decision-Support Environments

For DSS queries, corporate data is often consolidated into a separately designed environment, commonly called a *data warehouse*. A data warehouse stores business data for a company in a single, integrated relational database that can provide a historical perspective on information for DSS applications and ad-hoc queries.

Another approach to DSS operations, called a *data mart*, draws selected data from OLTP operations or a data warehouse to answer specific types of questions or to support a specific department or initiative within an organization.

DSS queries perform more complex and memory-intensive tasks than OLTP, often including scans of entire tables, manipulation of large amounts of data, multiple joins, and the creation of temporary tables. Such operations require large amounts of memory. Because of their complexity, DSS queries might not execute quickly.

Decision-support queries consume large quantities of non-CPU resources, particularly memory. The database server typically allocates large quantities of memory for the following SQL operators:

- Hash join
- Sort
- Group

Other factors also influence how the database server allocates resources to a query. Consider the following SELECT statement:

```
SELECT col1, col2 FROM table1 ORDER BY col1
```

If no indexes exist on **table1**, a sort is required, and the database server must allocate memory and temporary disk space to sort the query. However, if column **col1** is indexed, the database server can sometimes avoid performing the sort by scanning the table using the index and can automatically provide the ordering that the user requested, without consuming non-CPU resources.

Decision-support applications have the following characteristics:

- Complex queries that involve large amounts of data
- Large memory requirements
- Few users
- Periodic or ad-hoc requests
- Relatively long response times

Schemas for DSS Queries

Although DSS queries can use data accumulated through OLTP applications, that data can be queried most efficiently if it is loaded into a database with a schema created explicitly for queries instead of transaction processing.

A huge data warehouse might be further divided into data marts that are optimized for a particular kind of query, such as those that evaluate financial or marketing data. The process can also go in the other direction: existing data marts can be used to create data warehouses. Many users find that tests of DSS queries in smaller data marts gives them a better idea of how they might use a larger data warehouse.

Consider the following major customizable features of Extended Parallel Server when you create the schema for a data mart or data warehouse:

- Indexing techniques described in "Using Indexes" on page 13-13

  Bitmap indexes on columns with many duplicate values and Generalized-Key (GK) indexes on static tables can improve query processing speed. The database server can use more than one index on a table in processing a query.

- Fragmentation schemes, as described in Chapter 9, "Fragmentation Guidelines"

  Several table fragmentation schemes are available to increase data granularity for better fragment elimination in query processing and to improve the efficiency of parallel query processing.

- Memory-management and query-priority features for DSS queries, described in Chapter 12, "Resource Grant Manager"

  You can set configuration parameters and environment variables to specify how much shared memory is allotted to DSS queries and how much of that memory any single query can use.

  You can also assign a scheduling level for queries to help determine the order in which they are processed.

Dedicated Test Systems

You might decide to test queries or application designs on a system that does not interfere with production database servers. Even if your database server is used as a data warehouse, you might sometimes test queries on a separate system until you understand the tuning issues that are relevant to the query. However, testing queries on a separate system might distort your tuning decisions in several ways.

If you are trying to improve performance of a large query, one that might take several minutes or hours to complete, you can prepare a scaled-down database in which your tests can complete more quickly. However, be aware of these potential problems:

- The optimizer might make different choices in a small database than in a large one, even when the relative sizes of tables are the same. Verify that the query plan is the same in the real and the model databases.
- Execution time is rarely a linear function of table size. For example, sort time increases faster than table size, as does the cost of indexed access when an index goes from two to three levels. What appears to be a big improvement in the scaled-down environment can be insignificant when applied to the full database.

Therefore, any conclusions that you reach as a result of tests in the model database must be tentative until you verify them in the production database.

You can often improve performance by adjusting your query or data model. If you are using a multiuser system or a network, where system load varies widely from hour to hour, you might need to perform your experiments at the same time each day to obtain repeatable results. Start tests when the system load is consistently light so that you are truly measuring the impact of your query only.

Basic Approach to Performance Measurement and Tuning

Performance measurement and tuning involves consideration of two issues:

- Data management

  Managing data requires many constantly reevaluated decisions, such as where to store the data, how to access it, and how to protect it. These decisions affect performance.

- System workload

  The workload mix on your system also affects performance. For example, the optimal configuration for an database server used by 1,000 users who execute frequent short transactions is quite different from a configuration in which a few users make long and complicated decision-support queries. A database server that combines both kinds of use requires even more careful tuning of the configuration as well as transaction and query scheduling. Tuning your system for its best daily performance means striking a balance in the utilization of all system resources for all applications.

Early indications of a performance problem are often vague. Users might report these problems:

- The system seems sluggish.
- They cannot get all their work done.
- Response times for transactions are long or queries take longer than usual.
- The application slows down at certain times during the day.
- Transaction throughput is insufficient to complete the required workload.
- Transaction throughput decreases over a period of time.

To determine the nature of the problem, measure the actual use of system resources and evaluate the results.

To maintain optimum performance for your database applications, develop a plan for the following tasks:

- Making specific and regular measurements of system performance
- Making appropriate adjustments to maintain good performance
- Taking corrective measures whenever performance starts to degrade

Regular measurements can help you anticipate and correct performance problems. If you recognize problems early, you can prevent them from affecting your users significantly.

Informix recommends an iterative approach to performance tuning. If repeating the steps found in the following list does not produce the desired performance improvement, consider other causes of the problem. For example, insufficient hardware resources or problems within or between client applications, as mentioned in "Topics Beyond the Scope of This Manual" on page 1-36, also cause performance problems.

To tune performance

1. Establish performance objectives.
2. Measure resource utilization and database activity at regular intervals.
3. Identify symptoms of performance problems: disproportionate utilization of CPU, memory management, or disks.
4. Tune the operating-system configuration.
5. Tune the database server configuration.
6. Optimize chunk and dbspace configuration, including placement of logs, sort space, and disk space for temporary tables and sort files.
7. Optimize table placement, extent sizing, and fragmentation.
8. Improve index and disk space utilization.
9. Optimize background I/O activities, including logging, checkpoints, and page cleaning.
10. Schedule backup and batch operations for off-peak hours.
11. Consider optimizing the implementation of your database application.
12. Repeat steps 2 through 11.

Performance Goals

Many considerations go into establishing performance goals for the database server and the applications that it supports. For this reason, you need to state your performance goals and priorities clearly and consistently. Only then you can provide realistic and consistent expectations for the performance of applications.

The more precisely you define your performance goals, the easier it is to take steps to accomplish them.

Consider the following questions when you establish your performance goals:

- What is your top priority? Is it to maximize on-line transaction processing (OLTP) throughput, to minimize response time for specific decision-support queries, or to achieve the best overall mix?
- What kind of workload do you expect the database to support? What is the mix between simple transactions, extended decision-support queries, and other types of requests that the database server usually handles?
- At what point are you willing to trade transaction-processing speed against availability or the risk of data loss for a particular transaction?
- Is this database server instance used in a client / server configuration? If so, what are the networking characteristics that affect its performance?
- What is the maximum number of users that you expect to use the database server?
- What are your configuration resource limitations for memory, disk space, and CPU?
- Is this database server made up of multiple coservers? If so, what are the performance characteristics of the communication interface between coservers?

The answers to these questions can help you determine some realistic performance goals for your resources and your mix of applications.

Performance Measurements

The following measures describe the performance of a transaction-processing system:

- Resource utilization
- Throughput
- Response time
- Financial cost of a transaction

The following sections describe these measures.

Resource Utilization

The term *resource utilization* can have one of two meanings, depending on the context in which it is used. Resource utilization can refer to either:

- the amount of a certain resource that a particular operation requires or uses.

  The term is used in this first sense when you compare different approaches to perform a given task. For instance, if a sort operation requires 10 megabytes of disk space, its resource utilization is greater than another sort operation that requires only 5 megabytes of disk space.

- the current load on a particular system component.

  The term is used in this second sense when it refers, for instance, to the number of CPU cycles devoted to a particular query during a specific time interval.

For additional information about the ways that different load levels affect various system components, see "Resource Utilization and Performance" on page 1-25.

Throughput

Throughput is a measure of the amount of data that is processed in a given time period. For on-line transaction processing (OLTP) systems, throughput is typically measured in *transactions per second* (TPS) or *transactions per minute* (TPM). In any given installation, throughput depends on the following factors, among many:

- The specifications of the host computer
- The processing overhead in the software
- The layout of data on disk
- The degree of parallelism that both hardware and software support
- The types of transactions being processed
- The type of data being processed

Industry-Standard Throughput Benchmarks

Industrywide organizations such as the Transaction Processing Performance Council (TPC) provide standard benchmarks that allow reasonable throughput comparisons across hardware configurations and database servers. Informix is an active member in good standing of the TPC.

The TPC provides the following standardized benchmarks for measuring throughput:

- TPC-A

  This benchmark is used to compare simple on-line transaction-processing (OLTP) systems. It characterizes the performance of a simple transaction-processing system, emphasizing update-intensive services. TPC-A simulates a workload that consists of multiple user sessions connected over a network that involves significant disk I/O activity.

- TPC-B

  This benchmark is used to stress test peak database throughput. It uses the same transaction load as TPC-A but removes any networking and interactive operations to provide a best-case throughput measurement.

- TPC-C

  This benchmark is used for complex OLTP applications. It is derived from TPC-A and uses a mix of updates, read-only transactions, batch operations, transaction-rollback requests, resource contentions, and other types of operations on a complex database to provide a better representation of typical workloads.

- TPC-D

  This benchmark measures query-processing power in terms of completion times for very large queries. TPC-D is a decision-support benchmark built around a set of typical business questions phrased as SQL queries against large databases in the gigabyte or terabyte range.

Because every database application has its own particular workload, you cannot use TPC benchmarks to predict the throughput for your application. The actual throughput that you achieve depends largely on your application.

Throughput Measurement

The best way to measure throughput for an application is to include code in the application that logs the time stamps of transactions as they are committed.

If your application does not provide support for measuring throughput directly, you can obtain an estimate by tracking the number of COMMIT WORK statements that the database server logs during a given time interval. To obtain a listing of logical-log records written to log files, use the **onlog** utility, as described in the *Administrator's Reference*. Logged information also includes insert, delete, and update operations. However, you cannot obtain this information until it is written to a log file.

If you need more immediate feedback, you can use **onstat -p**, described in "The onstat Utility" on page 2-7, to gather an estimate. Before you run **onstat -p**, use the SET LOG statement to set the logging mode to unbuffered for the databases that contain tables of interest.

Response Time

Response time is specific to an individual transaction or query. Response time is typically the elapsed time from the moment that a user enters a command or activates a function until the application indicates that the command or function is complete. The response time for a typical database server application includes the following sequence of actions. Each action requires a certain amount of time to complete. The response time does not include the time that it takes for the user to think of and enter a query or request:

1. The application forwards a query to the database server.

   Then the database server performs the following tasks:

   a. Performs query optimization and determines if a query requires access to data that is fragmented across coservers

   b. Retrieves, adds, or updates the appropriate records and performs disk I/O operations directly related to the query on each participating coserver

   c. Performs any background I/O operations, such as logging, page cleaning, and so forth, that occur during the period in which the query or transaction is still pending

   d. Returns a result to the application

2. The application displays the information or issues a confirmation and then issues a new prompt to the user.

Figure 1-4 shows how these various intervals contribute to the overall response time when a transaction executes on a single coserver.

Figure 1-4
Components of the Response Time for a Single Transaction Executing on a Single Coserver

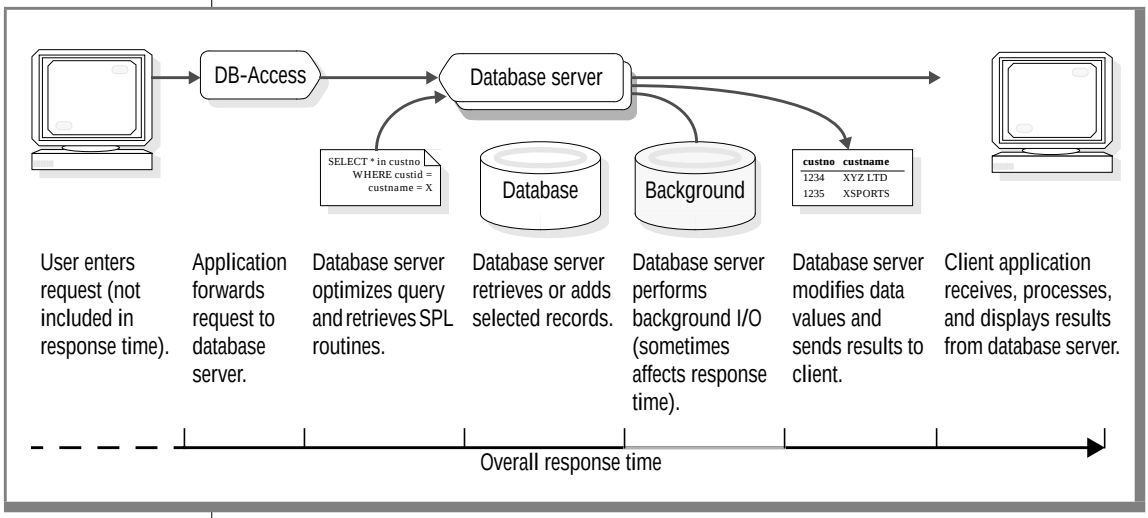
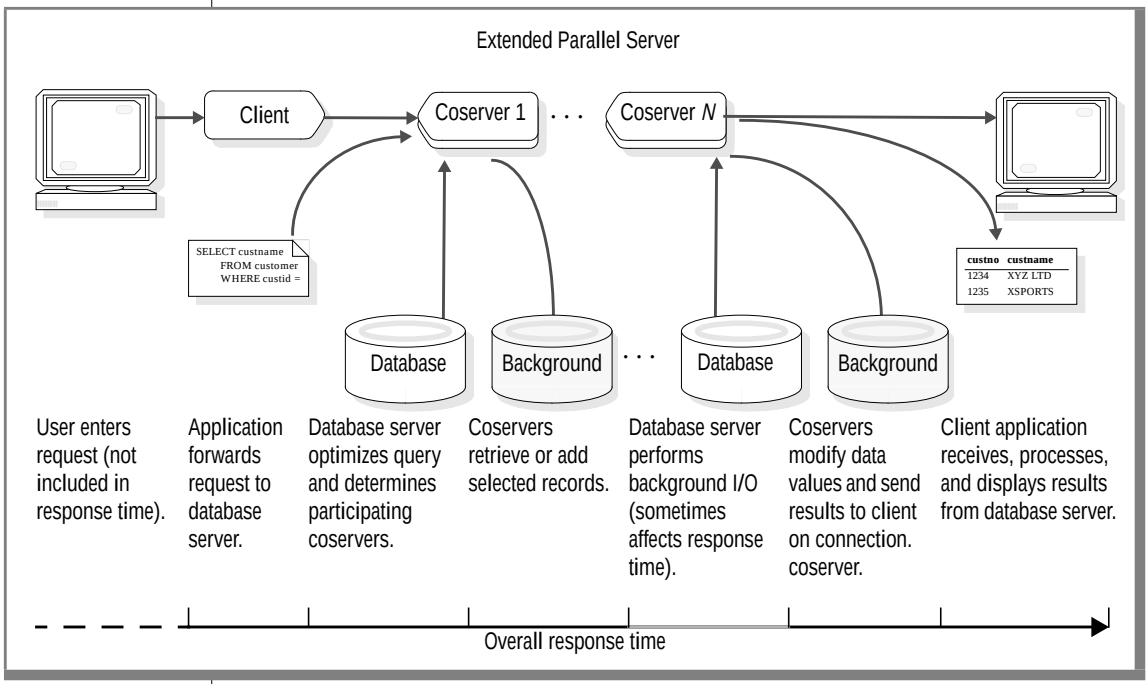


Figure 1-5 shows how these various intervals contribute to the overall response time on multiple coservers.

Figure 1-5
Components of the Response Time for a Single Transaction Executing on Multiple Coservers



Response Time and Throughput

Response time and throughput are related. The response time for an average transaction tends to decrease as you increase overall throughput. However, you can decrease the response time for a specific query, at the expense of overall throughput, by allocating a disproportionate amount of resources to that query. Conversely, you can maintain overall throughput by restricting the resources allocated to a large query.

The trade-off between throughput and response time becomes evident when you attempt to balance the ongoing need for high transaction throughput with an immediate need to perform a large decision-support query. The more resources you apply to the query, the fewer you have available to process transactions, and the larger the effect your query might have on transaction throughput. Conversely, the fewer resources you allow the query, the longer the query takes.

For suggestions about how to balance OLTP and DSS workloads in the database server, refer to "Managing Resources for DSS and OLTP Applications" on page 12-11.

Response-Time Measurement

You can use one of the following methods to measure response time for a query or application:

- Operating-system timing commands
- Operating-system performance monitor
- Timing functions within your application

Operating-System Timing Commands

Operating systems usually have a utility that you can use to time a command. You can often use this timing utility to measure the response times to SQL statements issued by a DB-Access command file.

If you have a command file that performs a standard set of SQL statements, you can use the **time** command on many systems to obtain an accurate time for those commands. For more information about command files, refer to the *DB-Access User's Manual*. The following example shows the output of the UNIX **time** command:

```
time commands.dba
...
4.3 real     1.5 user    1.3 sys
```

The **time** output lists the amount of elapsed time (real), the amount of time spent performing user routines, and the amount of time spent executing system calls. If you use the C shell, the first three columns of output from the C shell **time** command show the user, system, and elapsed times, respectively. In general, an application often performs poorly when the proportion of time spent on system calls exceeds one-third to one-half of the total elapsed time.

The **time** command gathers timing information about your application. You can use this command to invoke an instance of your application, perform a database operation, and then exit to obtain timing figures, as the following example illustrates:

```
time sqlapp
   (enter SQL command through sqlapp, then exit)
10.1 real    6.4 user    3.7 sys
```

When the application accesses data that resides on multiple coservers, use the **time** command on the connection coserver. Because all coservers work together to process a query and pass the result to the connection coserver, the amount of elapsed time can be measured on the connection coserver.

You can use a script to run the same test repeatedly, which allows you to obtain comparable results under different conditions. You can also obtain estimates of average response time by dividing the elapsed time for the script by the number of database operations that the script performs.

Operating-System Performance Monitor

Operating systems usually provide a performance monitor that you can use to measure response time for a query or process.

Timing Functions Within the Application

Most programming languages have a library function for the time of day. If you have access to the source code, you can insert pairs of calls to this function to measure the elapsed time between specific actions. For example, if the application is written in Informix ESQL/C, you can use the **dtcurrent()** function to obtain the current time. You can measure response time by calling **dtcurrent()** to report the time at the start of a transaction and again to report the time when the transaction commits.

In a system in which resources are shared among multiple processes, elapsed time does not always correspond to execution time. Most operating systems and C libraries contain functions that return the CPU time of a program.

Financial Cost of a Transaction

The financial cost per transaction is a measure that is usually used to compare overall operating costs among applications, database servers, or hardware platforms.

To measure the average financial cost per transaction

1. Calculate all the costs associated with operating an application. These costs can include the installed price of the hardware and software, operating costs (including salaries), and other expenses.
2. Project the total number of transactions and queries for the effective life of an application
3. Divide the total cost by the total number of transactions.

Although this measure is useful for management planning and evaluation, it is rarely relevant to achieving optimum database performance.

Resource Utilization and Performance

A typical OLTP application undergoes different demands throughout its various operating cycles. Peak loads during the day, week, month, and year, as well as the additional loads imposed by DSS queries or backup operations, can have a significant effect on a system that is running near capacity. You see these effects most clearly when you evaluate historical data from your own system.

To build up a profile of historical data, make regular measurements of the workload and performance of your system. Then you can predict peak loads and make appropriate comparisons between performance measurements at different points in your use cycle. This profile is important when you plan and evaluate performance improvements.

The database server provides several measurement tools, which are listed in Chapter 2, "Performance Monitoring." Your operating system also provides you with tools that measure the effect of performance on system and hardware resources, as noted under "Operating-System Timing Commands" on page 1-23.

Utilization is the percentage of time that a component is occupied, as compared with the percentage of time that the component is available for use. For instance, if a CPU processes transactions for a total of 40 seconds during a single minute, its utilization during that interval is 67 percent.

A *critical* resource is an overused resource or a resource whose utilization is disproportionate in comparison with that of other similar resources. For instance, you might consider a disk to be critical or overused when it has a utilization of 70 percent and all other disks on the system have 30 percent. Although 70 percent does not indicate that the disk is severely overused, you can improve performance if you balance I/O requests better across the coservers or the set of individual coserver disks.

How you measure resource utilization depends on the tools that your operating system provides to report system activity and resource utilization. When you identify a resource that seems overused, you can use database server performance-monitoring utilities to gather data and make inferences about the database activities that might account for the load on that component. You can adjust your database server configuration or your operating system to reduce those database activities or spread them among other components. In some cases, you might need additional hardware resources to resolve a performance bottleneck.

Resource Utilization

Whenever a system resource, such as a CPU or a particular disk, is occupied by a transaction or query, it is unavailable for processing other requests. Other pending requests must wait for the resources to become available before they can complete. When a resource is too busy to keep up with all requests, it becomes a bottleneck in the flow of activity through the system. The higher the percentage of time that the resource is occupied, the longer each operation must wait.

The major concern in performance tuning for a query is balance of processing across coservers and balanced resource use. Therefore, estimate resource use on every coserver.

You can use the following formula to estimate the *service time* for a specific request based on the overall utilization of the component that services the request. The expected service time includes the time spent both waiting for and using the resource in question. You can think of service time as that portion of the response time accounted for by a single component within your computer, as the following formula shows:

$$S \approx P/(1-U)$$

S is the expected service time.

P is the processing time that the operation requires once it obtains the resource.

U is the utilization for the resource.

As Figure 1-6 shows, the service time for a single component increases dramatically as the utilization increases beyond the 70 percent threshold. For instance, if a transaction requires one second of processing by a given component, you can expect it to take 2 seconds on a component at 50 percent utilization and 5 seconds on a component at 80 percent utilization. When demand for the resource reaches 90 percent, you can expect a transaction to take 10 seconds to make its way through that component.

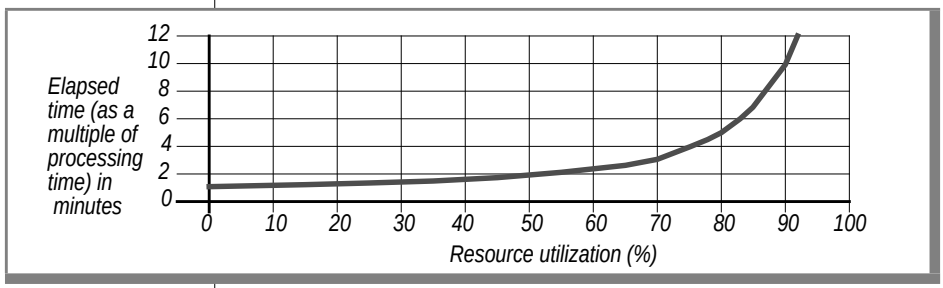


Figure 1-6 *Service Time for a Single Component as a Function of Resource Utilization*

If the average response time for a typical transaction soars from about 2 or 3 seconds to 10 seconds or more, users are certain to notice and complain.

It is important to monitor any system resource that shows a utilization of over 70 percent or any resource that exhibits the symptoms of overuse described in the following sections.

CPU Utilization

You can use the resource utilization formula given in the previous section to estimate the response time for a heavily loaded CPU on a coserver. However, high CPU utilization does not necessarily mean that there is a performance problem. The CPU performs all calculations needed to process transactions. The more transaction-related calculations it performs within a given period, the higher the throughput will be for that period. If throughput is high and seems to remain proportional to CPU utilization, high CPU utilization indicates that the computer is being used to fullest advantage.

On the other hand, if CPU utilization is high but transaction throughput does not keep pace, the CPU is either processing transactions inefficiently, or it is engaged in activity not directly related to transaction processing. If possible, eliminate the following extraneous activities:

- Large queries that might be scheduled at an off-peak time
- Unrelated application programs that might be performed on another computer

If the response time for transactions increases to such an extent that delays become unacceptable, the processor might be swamped; the transaction load might be too high for it to manage. Slow response time can also indicate that the CPU is processing transactions inefficiently or that CPU cycles are being diverted to internal housekeeping tasks, such as memory management or other activities.

When CPU utilization is high, a detailed analysis of the database server activities can reveal any sources of transaction processing inefficiency that might result from improper configuration. For information about how to analyze database server activity, refer to "Monitoring Database Server Resources" on page 2-9.

Memory Utilization

The formula to estimate memory utilization on performance is different from the formula for other system resources because the database server does not manage memory as a single resource component such as a CPU or disk.

Memory is managed as a collection of small components called *pages*. The size of a typical page in memory can range from 1 to 8 kilobytes, depending on your operating system. A computer with 64 megabytes of memory and a page size of 2 kilobytes contains approximately 32,000 pages.

For basic information about memory-management issues and the memory-management utility for your operating system, refer to the operating-system manuals.

The memory-management utility of the operating system allocates memory to a process in page-sized units. Many operating systems provide information about paging activity that includes the number of page scans performed, the number of pages removed from memory and written to the swap space on disk (*paged out*), and the number of pages brought in from the swap space (*paged in)*. Paging out is the critical factor because it occurs only when the memory manager cannot find free pages. A high rate of page scans is an early indication that memory utilization is becoming a bottleneck.

As demand for memory and the resulting paging activity increases, this activity increases until the CPU is almost completely occupied with paging calculations. A system in this condition is said to be *thrashing*. When a computer is thrashing, all useful work stops.

Paging-in activity does not accurately reflect the load on memory because pages for terminated processes are freed in place and reused. A high rate of paging in can result from a high rate of process turnover without significant performance effect.

Use the following formula to calculate the expected paging delay for a given CPU utilization level and paging rate:

```
P ≈ (C/(1-U)) * R * T
```

P is the paging delay.

C is the CPU service time for a transaction.

U is the CPU utilization (expressed as a decimal).

R is the paging-out rate.

T is the service time for the swap device.

Paging and CPU utilization are related. As paging increases, CPU utilization also increases, and these increases are compounded. If a paging rate of 10 per second accounts for 5 percent of CPU utilization, increasing the paging rate to 20 per second might increase CPU utilization by an additional 5 percent. Further increases in paging lead to even sharper increases in CPU utilization, until the expected service time for CPU requests becomes completely unacceptable.

Disk Utilization

Because each disk acts as a single resource, you can use the basic formula to estimate the service time:

```
S ≈ P/(1-U)
```

However, because transfer rates vary among disks, most operating systems do not report disk utilization directly. Instead, they report the number of data transfers per second (in operating-system memory-page-size units.) To compare the load on disks with similar access times, simply compare the average number of transfers per second.

If you know the manufacturer-provided access time for a given disk, you can calculate utilization for the disk by multiplying the average number of transfers per second by the access time. Depending on how the data is positioned on the disk, access times might vary from the manufacturer's rating. To account for this variability, Informix recommends that you add 20 percent to the access-time specification of the manufacturer.

The following example shows how to calculate the utilization for a disk with a 30-millisecond access time and an average of 10 transfer requests per second:

```
U =(A * 1.2) * X
  =(.03 * 1.2) * 10
  =.36
```

U is the resource utilization (in this case, a disk).

A is the access time that the manufacturer lists.

X is the number of transfers per second that your operating system reports.

Use the utilization to estimate the processing time at the disk for a transaction that requires a given number of disk transfers. To calculate the processing time at the disk, multiply the number of disk transfers by the average access time. Include an extra 20 percent to account for access-time variability:

```
P = D (A * 1.2)
```

P is the processing time at the disk.

D is the number of disk transfers.

A is the access time that the manufacturer lists.

For example, you can calculate the processing time for a transaction that requires 20 disk transfers from a 30-millisecond disk as follows:

```
P = 20 (.03 * 1.2)
  = 20 * .036
  = .72
```

Use the processing time and utilization values that you calculated to estimate the expected service time for I/O at the particular disk, as the following example shows:

```
S ≈ P/(1-U)
  = .72 / (1 - .36)
  = .72 / .64
  = 1.13
```

Factors That Affect Resource Use

Many factors affect how resources are used. This manual discusses some of these factors, but others are beyond its scope. Consider factors in these lists as you identify performance problems and adjust your system.

This manual discusses the following resource-use factors:

- Hardware resources

  Hardware resources include the CPU, physical memory, and disk I/O subsystems, as well as the hardware interconnects between coservers.

  Refer to "Resource Utilization and Performance," which begins on page 1-25.

  For additional information, refer to your operating-system and hardware-configuration manuals.

- The database server configuration

  Characteristics of your database server instance (such as number of coservers, size of resident and virtual shared-memory portions on each coserver, number of CPU VPs on each coserver, and so forth) play an important role in determining the capacity and performance of your applications.

  For more information on how database server configuration parameters affect performance, refer to Chapter 3, "Effect of Configuration on CPU Use," Chapter 4, "Effect of Configuration on Memory Use," and Chapter 5, "Effect of Configuration on I/O."

- Table types, table placement, and temporary dbspaces

  The table types that you use and the amount of space you allocate for temporary tables and sort space can affect the query processing time.

  For more information on table placement and disk space, refer to Chapter 6, "Table Performance."

- Table fragmentation

  Fragmenting tables across dbspaces and coservers can affect the speed at which the database server can locate data pages and transfer them to memory.

  For more information about fragmentation, refer to Chapter 9, "Fragmentation Guidelines."

- Space organization and extent sizing

  Fragmentation strategy and the size and placement of extents can affect the ability of the database server to scan a table rapidly. Avoid interleaved extents and allocate extents appropriately to accommodate growth of a table and prevent performance problems.

  Extent-related problems occur primarily with OLTP applications that insert and delete table rows.

  For more information, refer to "Managing Extents" on page 6-21 and Chapter 9, "Fragmentation Guidelines."

- Query efficiency

  Proper query construction and index use can decrease the load that any one application or user imposes. Large decision-support queries can take advantage of parallel execution to reduce the response time.

  For more information on the factors that affect query execution, refer to Chapter 10, "Queries and the Query Optimizer." For more information on how to improve query performance, refer to Chapter 13, "Improving Query and Transaction Performance." For more information on parallel execution of queries, refer to Chapter 11, "Parallel Database Query Guidelines."

- Scheduling background I/O activities

  Logging, checkpoints, page cleaning, and other operations (such as backups) can impose constant overhead and large temporary loads on the system. Schedule archive and batch operations for off-peak times whenever possible.

  For more information about the effect of background I/O activities, refer to Chapter 5, "Effect of Configuration on I/O."

This manual does not discuss the following resource-use factors:

- Operating-system configuration

  The database server depends on the operating system to provide low-level access to devices, process scheduling, interprocess communication, and other vital services.

  Your operating-system configuration directly affects how well the database server performs. The operating-system kernel takes up a significant amount of physical memory that the database server or other applications cannot use. However, you must reserve adequate kernel resources for use by the database server.

- Network configuration and traffic

  Applications that depend on a network for communication with the database server and systems that rely on data replication to maintain high availability are subject to the performance effects and constraints that the network imposes. Data transfers over a network are usually slower than data transfers from disk. Network delays can have a significant effect on the performance of the database server and application programs that run on the host computer.

- Application-code efficiency

  Application programs introduce their own load on the operating system, the network, and the database server. These programs can introduce performance problems if they make poor use of system resources, generate undue network traffic, or create unnecessary contention in the database server. Application developers must make proper use of cursors and locking levels to ensure good database server performance.

Maintenance of Good Performance

Performance is affected in some way by all system users: the database server administrator, the database administrator, the application designers, and the client application users.

The database server administrator usually coordinates the activities of all users to ensure that system performance meets overall expectations. For example, the operating-system administrator might need to reconfigure the operating system to increase the amount of shared memory. Bringing down the operating system to install the new configuration requires bringing the database server down. The database server administrator must schedule this downtime and notify all affected users when the system will be unavailable.

The database server administrator should:

- be aware of all performance-related activities that occur.
- educate users about the importance of performance, how performance-related activities affect them, and how they can assist in achieving and maintaining optimal performance.

The database administrator should pay attention to:

- how tables and queries affect the overall performance of the database server.
- how the distribution of data across coservers and disks affects performance.

Application developers should:

- design applications carefully to use the concurrency and sorting facilities that the database server provides, rather than attempt to implement similar facilities in the application.
- keep the scope and duration of locks to the minimum to avoid contention for database resources.
- include routines within applications that, when temporarily enabled at runtime, allow the database server administrator to monitor response times and transaction throughput.

Database users should:

- pay attention to performance and report problems to the database server administrator promptly.
- be courteous when they schedule large, decision-support queries and request as few resources as possible to get the work done.

Topics Beyond the Scope of This Manual

Important: *Although broad performance considerations also include reliability and data availability as well as improved response time and efficient use of system resources, this manual discusses only response time and system resource use. For discussions of improved database server reliability and data availability, see information about failover, mirroring, high availability, and backup and restore in the "Administrator's Guide" and the "Backup and Restore Guide."*

Attempts to balance the workload often produce a succession of moderate performance improvements. Sometimes the improvements are dramatic; however, in some situations a load-balancing approach is not enough. The following types of situations might require measures beyond the scope of this manual:

- Application programs that require modification to make better use of database server or operating-system resources
- Applications that interact in ways that impair performance
- A host computer that might be subject to conflicting uses
- A host computer with inadequate capacity for the evolving workload
- Network performance problems that affect client/server or other applications

No amount of database tuning can correct these situations. Nevertheless, they are easier to identify and resolve when the database server is configured properly.

Chapter

2

Performance Monitoring

In This Chapter

This chapter explains the performance monitoring tools that you can use and how to interpret the results of performance monitoring. The descriptions of the tools can help you decide which tools to use for the following purposes:

- To create a performance history
- To monitor database server resources
- To monitor sessions and queries

The kinds of data that you need to collect depend on the kinds of applications you run on your system. The causes of performance problems on systems used for on-line transaction processing applications (OLTP) are different from the causes of problems on systems that are used primarily for DSS query applications. Systems with mixed use present a performance-tuning challenge and require a sophisticated analysis of performance problems.

Evaluating Your Current Configuration

Before you adjust your database server configuration, you should evaluate the performance of its current configuration.

If database applications perform well enough to satisfy user expectations, do not make frequent adjustments, even if those adjustments might produce a theoretical improvement in performance. Changing the database server configuration might interrupt users' work. Altering some features requires you to bring down the database server. Making configuration adjustments might degrade performance or cause other negative side effects.

As long as users are reasonably satisfied, take a gradual approach when you reconfigure the database server. If possible, evaluate configuration changes in a test instance of the database server before you change the configuration of your production system.

The utilities and methods that are described in this chapter can help you create a performance history of your current database server configuration. Use the information in the performance history to identify the performance bottlenecks in your system.

Creating a Performance History

Begin scheduled monitoring of resource use as soon as you set up your database server and begin to run applications on it. To accumulate data for performance analysis, use operating-system and command-line utilities in scripts or batch files and write the output to log files. For information about using command-line utilities, see "Command-Line Utilities" on page 2-6 and "Operating-System Tools" on page 2-5.

Importance of a Performance History

To build up a performance history and profile of your system, take frequent regular snapshots of resource-utilization information:

- Chart the CPU utilization and paging-out rate for each coserver, and the I/O transfer rates for the disks on your system to identify peak-use levels, peak-use intervals, and heavily loaded coserver components.
- Monitor fragment use to determine whether your fragmentation scheme is correct.
- Monitor other resource use as appropriate for your database server configuration and the applications that run on it.

If you have history information on hand, you can begin to track down the cause of problems as soon as users report slow response or inadequate throughput. If history information is not available, you must start tracking performance after a problem arises, and you cannot tell when and how the problem began. Trying to identify problems after the fact significantly delays resolution of a performance problem.

Choose tools from those described in the following sections, and create jobs that build up a history of disk, memory, I/O, and other database server resource use. To help you decide which tools to use to create a performance history, the output of each tool is described briefly.

Tools That Create a Performance History

When you monitor database server performance, you use tools from the host operating system and run command-line utilities at regular intervals from scripts or batch files. You can also use graphical interface tools to monitor critical aspects of performance as queries and transactions are performed.

Operating-System Tools

The database server relies on the operating system of the host computer to provide access to system resources such as the CPU, memory, and various unbuffered disk I/O interfaces and files. Each operating system has its own set of utilities for reporting how system resources are used. Different implementations of some operating systems have monitoring utilities with the same name but different options and informational outputs.

You might be able to use some of the following typical UNIX operating-system utilities to monitor resources.

UNIX Utility	Description
vmstat	Displays virtual-memory statistics.
iostat	Displays I/O utilization statistics. This utility is useful for measuring and comparing disk activity. All disks should be equally used.
sar	Displays a variety of resource statistics.
ps	Displays active process information, including memory assigned to each process. This utility is useful for finding runaway or problem processes that have accumulated a large amount of CPU time or a large amount of memory.

For details on how to monitor your operating-system resources, consult the reference manual or your system administration guide.

To capture the status of system resources at regular intervals, use scheduling tools that are available with your host operating system (for example, **cron**) as part of your performance monitoring system.

Your operating system might also provide graphical monitoring tools that you can use to monitor resources dynamically across coservers. For example, **3dmon** displays a variety of resource statistics in a three-dimensional graph.

Command-Line Utilities

Informix database servers provide command-line utility programs to monitor performance. In Extended Parallel Server, you can use these utility programs as arguments to the **xctl** utility to collect performance data from all coservers in the database server, or from specified coservers, and display it in a single output report. You can also run the command-line utilities on an individual coserver to retrieve performance information for that coserver only.

To capture information about configuration and performance across all coservers, use the **xctl** utility with the **onstat** options described in "Monitoring Database Server Resources" on page 2-9. You can also use **xctl** with **onlog** to examine the logical logs, as the *Administrator's Reference* describes.

You can also use SQL SELECT statements to query the system-monitoring interface (SMI) from within your application.

The SMI tables are a collection of tables and pseudo-tables in the **sysmaster** database that contain dynamically updated information about the operation of the database server. The tables are constructed in memory but are not recorded on disk. To query SMI tables at regular intervals, use **cron** jobs and SQL scripts with DB-Access. For information about SQL scripts, refer to the *DB-Access User's Manual*.

Tip: *The SMI tables are different from the system catalog tables. System catalog tables contain permanently stored and updated information about each database and its tables. For information about SMI tables, refer to the "Administrator's Reference." For information about system catalog tables, refer to the "Informix Guide to SQL: Reference."*

The xctl Utility

Use **xctl** (execute utilities across coservers) to capture performance-related information for all coservers in your database server.

Use **xctl** with **onstat** and its arguments to display information about the current status and activities of the database server on all or specified coservers. To display a complete list of the **onstat** options, use **onstat - -**. Use **xctl** with **onlog** to display all or part of the logical log or to check the status of the logical logs.

The onstat Utility

You can run **onstat** either as an argument to **xctl** or alone from the command line on the coserver where you are logged in. When you run **onstat** without **xctl**, you see information about the local coserver only.

Enter **xctl onstat** without any arguments to see the current status of the database server, how long it has been running, a list of user threads, and a summary of current activity.

Tip: *Profile information displayed by* ***onstat*** *commands, such as* ***onstat -p,*** *accumulates from the time the database server was initialized. To clear performance profile statistics so that you can create a new profile, run* ***xctl onstat -z****. If you use* ***xctl onstat -z*** *to reset statistics for a performance history or appraisal, make sure that other users do not also enter the command at different intervals.*

Use the following **xctl onstat** options to display general performance-related information.

Option	Description
xctl onstat -p	Displays a performance profile that includes the number of reads and writes, the number of different ISAM calls of all categories, the number of times that a resource was requested but was not available, and other miscellaneous information.
xctl onstat -x	Displays information about transactions, including the thread identifier of the user who owns the transaction.
xctl onstat -u	Displays a user activity profile that provides information about user threads, including the thread owner's session ID and login name.
xctl onstat -R	Displays each LRU queue number and type and the total number of queued and dirty buffers. For the performance implications of LRU queue statistics, see "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-28.
xctl onstat -F	Displays page-cleaning statistics that include the number of writes of each type that flushes pages to disk.
xctl onstat -g	Requires an additional argument that specifies the information to be displayed, such as **xctl onstat -g mem** to monitor memory on all coservers.

For more information about options that provide performance-related information, see "Monitoring Database Server Resources" on page 2-9. For detailed information about these command-line utilities, refer to the *Administrator's Reference*.

The onlog Utility

In addition to the performance history that you build up, you can use the **onlog** utility to display contents of logical-log files. Use **xctl** with **onlog** to display logical-log contents on all coservers.

Logical-log records might help you identify a problem transaction or assess transaction activity that corresponds to a period of high utilization, as indicated by your performance history. Records in the logical log might help you identify performance problems that are associated with specific users, tables, or transactions.

The **onlog** utility displays all or selected portions of the logical log. This command can display information from selected log files, the entire logical log, or an archive tape of previous log files. The **onlog** utility displays all or selected portions of the logical log. This command can display information from selected log files, the entire logical log, or an archive tape of previous log files.

To check the status of logical-log files, use **onstat -l**.

Warning: *Use **onlog** to read logical-log files from backup tapes only. While you use **onlog** to read logical-log files on disk before they have been backed up, the database server locks the log files and stops database activity for all sessions.*

Monitoring Database Server Resources

Monitor specific database server resources to identify performance bottlenecks and potential trouble spots, and improve resource use and response time. You can monitor threads, the network, and virtual processors.

One of the most useful commands for monitoring system resources is **onstat -g** and its many options. You can run **onstat -g** on individual coservers or add it as an argument to the **xctl** command to display information about all coservers. "Using Command-Line Utilities to Monitor Queries" on page 12-30 and "Monitoring Fragmentation" on page 9-65 contain many **onstat -g** examples.

Use the following **xctl onstat -g** options to monitor threads.

onstat -g Option	Description
xctl onstat -g act	Lists active threads.
xctl onstat -g ath	Lists all threads. The **sqlexec** threads represent portions of client sessions; the **rstcb** value corresponds to the **user** field that the **onstat -u** command displays. For more information and sample output, see "Monitoring User Threads and Transactions" on page 12-43.
xctl onstat -g rea	Lists ready threads.
xctl onstat -g sle	Lists all sleeping threads.
xctl onstat -g sts	Lists maximum and current stack use per thread.
xctl onstat -g tpf *tid*	Displays a thread profile for *tid*. If you enter a *tid* of `0`, this argument displays profiles for all threads.

Use the following **xctl onstat -g** options to monitor the network.

onstat -g Option	Description
xctl onstat -g ntd	Lists network statistics by service.
xctl onstat -g ntt	Lists network user times.
xctl onstat -g ntu	Lists network user statistics.
xctl onstat -g qst	Lists queue statistics.

Use the following **xctl onstat -g** options to monitor virtual processors.

onstat -g Option	Description
xctl onstat -g sch	Lists the number of semaphore operations, spins, and busy waits for each VP.
xctl onstat -g spi	Lists longspins, which are spin locks that virtual processors have spun more than 10,000 times in order to acquire a lock. To reduce longspins, reduce the number of virtual processors or reduce the load on the computer. On some platforms, you can use the *no-age* or *processor affinity* features.
xctl onstat -g wst	Displays wait statistics.
xctl onstat -g glo	Lists global multithreading information. This listing includes CPU-use information about virtual processors, the total number of sessions, and other multithreading global counters.

Monitoring Sessions

Use the following **xctl onstat -g** options to monitor sessions.

onstat -g Option	Description
xctl onstat -g ses	Lists summaries of all sessions.
xctl onstat -g ses *session id*	Lists session information by session ID. If you omit *session id*, this option displays one-line summaries of all active sessions.
xctl onstat -g sql *session id*	Lists SQL information by session. If you omit *session id*, this option displays summaries of all sessions.
xctl onstat -g stk *thid*	Lists stack information by thread. The *thid* variable is the value in the **tid** field that appears in the output of the **onstat -g ses** commands.

For examples and discussions of session-monitoring command-line utilities, see "Using Command-Line Utilities to Monitor Queries" on page 12-30.

Monitoring Memory Use

Use the following arguments to **xctl onstat -g** to display memory-utilization information. For overall memory information on coservers, omit *table name*, *pool name*, or *session id* from the commands that permit those optional parameters.

onstat -g Option	Description
xctl onstat -g ffr *pool name* | *session id*	Lists free fragments for a pool of shared memory.
xctl onstat -g dic *table name*	Displays one line of information for each table that is cached in the shared-memory dictionary. If you enter a specific table name as a parameter, this option displays internal SQL information about that table, including constraints and index descriptors and fragments.
xctl onstat -g iob	Displays large private buffer use by I/O virtual-processor class.
xctl onstat -g mem *pool name* | *session id*	Displays memory statistics for the pools associated with a session. If you do not provide an argument, this option displays pool information for all sessions.
xctl onstat -g nsc *client id*	Displays shared-memory status by client ID. If you omit *client id*, this option displays all client status areas.
xctl onstat -g nsd	Displays network shared-memory data for poll threads.
xctl onstat -g nss *session id*	Displays network shared-memory status by session ID. If you omit *session id*, this option displays all session status areas.
xctl onstat -g seg	Displays shared-memory segment statistics. This option shows the number and size of all attached segments.
xctl onstat -g ufr *pool name* | *session id*	Displays allocated pool fragments by user or session.

Monitoring Data Distribution and Table Fragmentation Use

Extended Parallel Server uses fragmentation of data across coservers to achieve full parallel processing. You can monitor the following aspects of fragmentation:

- Data distribution over fragments
- I/O request balancing over fragments
- Status of chunks that contain fragments

Monitoring Data Distribution over Fragments

The database server administrator can monitor data distribution over table fragments. Because the unit of disk storage for a fragment is a tblspace, you monitor tblspaces to monitor fragmentation disk use.

If the goal of fragmentation is improved OLTP response time, data should be distributed evenly over the fragments. To determine an appropriate fragmentation method and scheme, examine the SQL statements that applications use to access the data.

Balancing I/O Requests over Fragments

The database server administrator must monitor I/O request queues for data in table fragments. When I/O queues are unbalanced and some fragments are used more than others, the database server administrator should work with the database administrator to tune the fragmentation strategy.

Use the **onutil** CHECK SPACE command to monitor chunks and extents. The **onutil** CHECK SPACE command provides the following information:

- Chunk size
- Number of free pages
- Tables within the chunk

This information allows you to monitor chunk I/O activity and track the disk space that chunks use.

Querying System Catalog Tables for Table-Fragment Information

Query the **sysfragments** and **systables** system catalog tables to get information about all tblspaces that hold a fragment and the table to which each fragment belongs.

After you obtain the names of tables that have fragments stored in specific chunks, keep the information for later reference. If you alter tables, add tables, or otherwise change the physical layout of the database, retrieve the table information again.

For more information, refer to "Monitoring Fragmentation" on page 9-65.

Monitoring Chunks

Use the following command-line options to monitor chunks. For any of these commands, you can use the **xctl -c** *n* option to monitor chunks on a single specified coserver displays.

onstat -g Option	Description
xctl onstat -d	Displays the address of the chunk, its number and associated dbspace number, offset, size in pages, number of free pages, number of blobpages, and the pathname of the physical device.
xctl onstat -D	Displays the same information as **xctl onstat -d** but replaces the free page information with columns that list the number of pages read from and written to the chunk.
xctl onstat -g iof	Displays the number of reads from and writes to each chunk.
xctl -c *n* **onstat -g ppf**	Displays the number of read and write calls for each fragment on the coserver that is specified after the **-c** option.

Monitoring Data Flow Between Coservers

To make sure that the database sever is properly balanced across coservers, you can monitor statistics that indicate how packets are transmitted across the high-speed interconnect and how often the database server checks the interconnect but has no work to transmit.

Use the following command-line options to monitor the high-speed interconnect. For sample output and detailed information about these data-flow monitoring commands, see "Monitoring Data Flow Between Coservers" on page 12-46.

onstat -g Option	Description
xctl onstat -g dfm	Displays data-flow manager information, such as sender, receiver, and global-packet information
xctl onstat -g xmf	Displays messaging information, such as poll, memory information, end-point, and overall cycle statistics

Monitoring Sessions and Queries

One of the most important overall factors in efficient application planning is avoiding database server imbalance, or *skew*.

To analyze database server efficiency, use the database server utility programs that are listed in the following sections. You can also use UNIX operating-system utility programs such as **sar** and **3dmon** to monitor disk I/O wait times, service times, and queue lengths. The operating-system utilities available depend on your platform.

Monitoring Sessions

Some **xctl onstat** commands are particularly useful for monitoring sessions.

Use the **xctl onstat -t** option to display the following information about active tblspaces:

- The number of open tblspaces
- Each active tblspace
- The number of user threads currently accessing the tblspace
- Whether the tblspace is busy or dirty
- The number of allocated and used pages in the tblspace
- The number of noncontiguous extents allocated

For information about how to associate the tblspace information with the table name, refer to the instructions in "xctl -c n onstat -g ppf" on page 9-70.

Monitoring Queries

You can monitor DSS queries with the following tools:

- The SQL statement SET EXPLAIN
- **onstat -g** options

Using SET EXPLAIN

Use SET EXPLAIN to produce the query plan that the optimizer creates for an individual query. For more information, refer to "Using SET EXPLAIN to Analyze Query Execution" on page 12-24.

Using onstat -g Options

Use the following **onstat-g** options to monitor DSS application sessions and queries.

onstat -g Option	Description
xctl onstat -g xmp	Displays active query segments and SQL operators.
onstat -g rgm	Displays RGM information. Do not use the **xctl** utility with the **onstat -g rgm** option. You can execute **onstat -g rgm** only on **coserver 1**.
xctl onstat -g xqp *qryid*	Displays query plans about a specified query.
xctl onstat -g xqs *qryid*	Displays statistics for SQL operators.
xctl onstat -u	Displays user threads and transactions on all coservers.
xctl onstat -g *session id*	Displays the resources allocated for and used by a session.
xctl onstat -g sql	Displays SQL information by session.

For more information about these monitoring commands, including output samples, see "Using Command-Line Utilities to Monitor Queries" on page 12-30.

In addition, **xctl onstat -t**, described on page 2-16, and **xctl onstat -d**, described on page 2-14, can provide information about DSS query sessions. Interconnect and related statistics appear in the output of **xctl onstat -g dfm** and **xctl onstat -g xmf**, which are listed in "Monitoring Data Flow Between Coservers" on page 2-15.

Performance Problems Not Related to the Database Server

Not all performance problems are database server problems. This section describes some performance problems related to other factors in your system:

- When performance problems are associated with backup operations, examine the transfer rates for tape drives. You might decide to change table layout or fragmentation to reduce the effect of backup operations. For information about disk layout and table fragmentation, refer to Chapter 6, "Table Performance," and Chapter 9, "Fragmentation Guidelines."
- For client / server configurations, consider network performance and availability. Evaluating network performance is beyond the scope of this manual. For information on how to monitor network activity and improve network availability, see your network administrator or refer to the documentation for your network system.
- The coservers that make up the database server communicate through a high-speed interconnect in massively parallel processor (MPP) or loosely clustered systems. The interconnect is a potential source of performance problems. For information on how to monitor communication between the coservers, refer to "Monitoring Data Flow Between Coservers" on page 2-15.

Routine system-maintenance jobs might conflict with database server use of the system. If system maintenance jobs are scheduled at regular times and database server performance declines at those times, you might suspect such a conflict.

Chapter

3

Effect of Configuration on CPU Use

In This Chapter

This chapter contains a summary of the operating-system and database server configuration parameters and environment variables that affect CPU use.

It describes the parameters that most directly affect CPU use and explains how to set them. Where possible, this chapter also provides suggested settings or considerations for different types of workloads.

For details about the syntax of the database configuration parameters, refer to the *Administrator's Reference*.

UNIX Parameters That Affect CPU Use

As described in "Documentation Notes, Release Notes, Machine Notes" on page 12, your database server distribution includes a computer-specific file that contains recommended values for UNIX configuration parameters. Compare the values in the machine notes file with your current operating-system configuration.

The following UNIX parameters affect CPU use:

- Semaphore parameters
- Parameters that set the maximum number of open file descriptors
- Memory configuration parameters

UNIX Semaphore Parameters

Semaphores are kernel resources with a typical size of 1 byte each. Allocate semaphores for the database server in addition to any that you allocate for other software packages.

On each coserver, allocate 1 UNIX semaphore for each virtual processor (VP), 1 for each user who connects to the database server through shared memory, 6 for database server utilities, and 16 for other purposes.

Tip: *For best performance, Informix recommends that you allocate enough semaphores for twice as many* ***ipcshm*** *connections as you expect. Informix also recommends that you use the NETTYPE parameter to configure the database server poll threads for this doubled number of connections. For information on configuring poll threads, refer to* *"NETTYPE" on page 3-14**. For a description of poll threads, refer to the "Administrator's Guide."*

Because some database server utilities use shared-memory connections, you configure a minimum of two semaphore sets for each coserver: one for the initial set of VPs and one for the shared-memory connections that database server utilities use. The SEMMNI operating-system configuration parameter usually specifies the number of semaphore sets that are allocated in the operating system. For information on how to set semaphore-related parameters, refer to the configuration instructions for your operating system.

The SEMMSL operating-system configuration parameter usually specifies the maximum number of semaphores per set. Set this parameter to at least `100`.

Some operating systems require that you configure a maximum total number of semaphores across all sets, which the SEMMNS operating-system configuration parameter typically provides. Use the following formula to calculate the total semaphores required for each instance of the database server:

```
SEMMNS = init_vps + added_vps + (2 * shmem_users) + concurrent_utils
```

init_vps is the number of VPs that are initialized with the database server. This number includes CPU, PIO, LIO, AIO, FIF, SHM, TLI, SOC, and ADM VPs. (For a description of these VPs, see the *Administrator's Guide*.) The minimum value for this term is 15.

added_vps is the number of VPs that you intend to add dynamically.

shmem_users is the number of shared-memory connections that you allow for this instance of the database server.

concurrent_utils is the number of concurrent database server utilities that can connect to this instance. Informix suggests that you allow for a minimum of six utility connections: two for **onbar** and four for other utilities such as **onstat** and **onutil**.

If you use software packages that require semaphores in addition to those that the database server needs, add these semaphores to the total calculated by the previous formula when you set the SEMMNI configuration parameter. Set the SEMMSL configuration parameter for the largest number of semaphores per set that any of your software packages require. For systems that require the SEMMNS configuration parameter, multiply SEMMNI by the value of SEMMSL to calculate an acceptable value.

UNIX File-Descriptor Parameters

Some operating systems require you to specify a limit on the number of file descriptors that a process can have open at any one time. You specify this limit with an operating-system configuration parameter, usually NOFILE, NOFILES, NFILE, or NFILES. The number of open-file descriptors that each instance of the database server needs is determined by the number of chunks in the database, the number of VPs that the database server runs, and the number of network connections that the database server instance must support. Network connections include all except those specified as the **ipcshm** connection type in either the **sqlhosts** file or a NETTYPE database server configuration entry.

Use the following formula to calculate the number of file descriptors that your instance of the database server requires:

```
NFILES = (chunks * NUMAIOVPS) + NUMCPUVPS + net_connections
```

chunks is the number of chunks to be configured.

net_connections is the number of network connections (other than **ipcshm**) that your instance of the database server supports.

Each open file descriptor is about the same length as an integer within the kernel. Allocating extra file descriptors is an inexpensive way to allow for growth in the number of chunks or connections on your system.

UNIX Memory-Configuration Parameters

The configuration of memory in the operating system can affect other resources, including CPU and I/O. Insufficient physical memory for the overall system load can lead to thrashing, which is described in Chapter 4, "Effect of Configuration on Memory Use." Insufficient memory for the database server can result in excessive buffer-management activity. For more information on configuring memory, refer to "Configuring Shared Memory" on page 4-8.

Configuration Parameters and Environment Variables That Affect CPU Use

Several database server configuration parameters and environment variables affect CPU use. Your hardware configuration determines some parameter settings; others depend on the work load of your database server. The following sections discuss each parameter and environment variable that affects CPU use.

Some configuration parameters are set differently for uniprocessor and multiprocessor nodes. For uniprocessor coserver nodes, set the following configuration parameters.

Parameter	Value
MULTIPROCESSOR	0
NUMAIOVPS	2
NUMCPUVPS	1
SINGLE_CPU_VP	1

For symmetric multiprocessor (SMP) coserver nodes, refer to your machine notes for appropriate settings for these parameters as well as for the NOAGE parameter. In some SMP systems, you can also set the following configuration parameters to bind processes to specific CPUs:

- AFF_NPROCS
- AFF_SPROC

The following environment variable and configuration parameters also affect CPU use. You can adjust them for performance tuning on any hardware configuration:

- DS_MAX_QUERIES
- MAX_PDQPRIORITY
- NETTYPE
- PDQPRIORITY
- PSORT_NPROCS

The following sections describe the performance effects and considerations that are associated with these parameters. For more information about database server configuration parameters, refer to your *Administrator's Reference*.

NUMCPUVPS, MULTIPROCESSOR, and SINGLE_CPU_VP

These configuration parameters specify the number of CPU VPs that can run on each coserver and how locking is performed if you are running more than one CPU VP.

The number of CPU VPs is an important factor in determining the number of scan threads for a query. Queries perform best when the number of scan threads is a multiple or factor of the number of CPU VPs so that each VP can be assigned the same amount of work.

Adding or removing a CPU VP can improve performance for a large query because it produces an equal distribution of scan threads among CPU VPs. For instance, if you have 6 CPU VPs and scan 10 table fragments, you might see a faster response time if you reduce the number of CPU VPs to 5, which divides evenly into 10. You can use **xctl onstat -g ath** to monitor the number of scan threads per CPU VP across coservers or use **xctl onstat -g ses** to focus on a particular session.

Use **xctl onstat -g rea** to monitor ready queues across coservers. To make sure that you are using the system CPU resources fully, note the size of the thread ready queues and the number of CPU VP threads in the **onstat -g rea** output. You might improve performance by adding a CPU VP if the CPU VP thread queue is a consistent length and CPU idle time or I/O wait time is available.

NUMCPUVPS

The NUMCPUVPS configuration parameter specifies the number of CPU VPs that the database server brings up initially on each coserver.

Do not allocate more CPU VPs than the number of CPUs available to service them:

- For uniprocessor coserver nodes, Informix recommends that you use one CPU VP.
- For multiprocessor coserver nodes with four or more CPUs that are primarily used as database servers, Informix recommends that you set NUMCPUVPS to the total number of processors.
- For multiprocessor systems that are not primarily used to support database servers, you can start with somewhat fewer CPU VPs to allow for other activities on the system and then gradually add more if necessary.

For dual-processor systems, you might improve performance by running with two CPU VPs.

The number of CPU VPs on each coserver should be equal to or less than the number of physical CPUs. The database server ensures optimal use of available CPU resources by each CPU VP. If there is enough work to be done, each CPU VP uses all of the resources provided by one physical CPU. Specifying more CPU VPs than physical processors can result in significant contention for CPU resources.

If you specify more CPU VPs than CPUs, you probably will not notice any performance degradation if the system is not heavily loaded and one or more of the CPU VPs is idle most of the time. Nevertheless, as activity on the system increases, performance will degrade noticeably at some point.

MULTIPROCESSOR

If you are running multiple CPU VPs, set the MULTIPROCESSOR configuration parameter to `1`. When you set MULTIPROCESSOR to `1`, the database server performs locking appropriately for a multiprocessor. If you are not running multiple CPU VPs, set this parameter to `0`.

SINGLE_CPU_VP

If you are running only one CPU VP, set the SINGLE_CPU_VP configuration parameter to 1. Otherwise, set this parameter to 0.

Important: *If you set the SINGLE_CPU_VP parameter to* 1*, the value of the NUMCPUVPS parameter must also be* 1*. If NUMCPUVPS is greater than* 1*, the database server fails to initialize and displays the following error message:*

```
Cannot have 'SINGLE_CPU_VP' non-zero and 'NUMCPUVPS' greater than 1
```

When the SINGLE_CPU_VP parameter is set to 1, you cannot add CPU VPs while the database server is in on-line mode.

NOAGE

To disable process priority aging for the database server CPU VPs on operating systems that support this feature, set NOAGE to 1. When process priority aging is disabled, critical database server processes can continue to run at high priority.

For information on whether your operating system supports this feature, refer to the machine notes file. If your operating system does not support this feature, consider using the **renice** command or another UNIX operating system utility to increase the priority level for the database server.

AFF_NPROCS and AFF_SPROC

On multiprocessor host computers that support processor affinity, the database server supports automatic binding of CPU VPs to processors. For information on whether your version of the database server supports processor affinity, refer to the machine notes file.

For information about using AFF_NPROCS and AFF_SPROC to bind CPU VPs to processors, see "Setting AFF_NPROCS and AFF_SPROC" on page 3-11.

Using Processor Affinity

Processor affinity can distribute the computation effect of CPU VPs and other processes. On coserver nodes that are dedicated to the database server, assigning CPU VPs to all but one of the CPUs achieves maximum CPU use. On coserver nodes that support both database server and client applications, you can use the operating system to bind applications to certain CPUs. You can bind the remaining CPUs with the AFF_NPROCS and AFF_SPROC configuration parameters.

For coserver nodes that run both DSS and OLTP client applications, you might bind asynchronous I/O (AIO) VPs to the same CPUs to which you bind other application processes through the operating system. In this way, you isolate client applications and database I/O operations from the CPU VPs. This isolation can be especially helpful when client processes are used for data entry or other operations that wait for user input. Because AIO VP activity usually comes in quick bursts followed by idle periods of waiting for the disk, client and I/O operations can often be interleaved without unduly affecting each other.

To prevent all coservers from using the same CPUs on nodes where you run more than one coserver, you might set AFF_SPROC in the coserver-specific sections of the ONCONFIG file. Processor affinity also makes it easier to use operating-system tools such as **mpstat** to monitor the coserver load balance.

Binding a CPU VP to a processor does not prevent other processes from running on that processor. Processes that you do not bind to a CPU are free to run on any available processor. On a computer that is dedicated to the database server, you can leave AIO VPs free to run on any processor, which reduces delays on database operations that are waiting for I/O. Increasing the priority of AIO VPs can further improve performance by ensuring that data is processed quickly as soon as it arrives from disk.

Setting AFF_NPROCS and AFF_SPROC

The AFF_NPROCS and AFF_SPROC parameters specify the number of CPU VPs to bind with processor affinity and the processors to which those VPs are bound. When you assign a CPU VP to a specific CPU, the VP runs only on that CPU, but other processes can also run on that CPU.

Set the AFF_NPROCS parameter to the number of CPU VPs. (See "NUMCPUVPS, MULTIPROCESSOR, and SINGLE_CPU_VP" on page 3-8.) Do not set AFF_NPROCS to a number that is greater than the number of CPU VPs.

You can set the AFF_SPROC parameter to the number of the first CPU on which to bind a CPU VP. The database server assigns CPU VPs to CPUs serially, starting with that CPU. To avoid assigning a certain CPU, set AFF_SPROC to 1 plus the number of that CPU.

You usually alter the value of this parameter only when you know that a certain CPU has a specialized hardware or operating-system function (such as interrupt handling), and you want to avoid assigning CPU VPs to that processor.

NUMAIOVPS

The NUMAIOVPS configuration parameter specifies the number of AIO VPs that the database server brings up initially on a coserver node. If your operating system does not support kernel asynchronous I/O (KAIO), the database server uses AIO VPs to manage all database I/O requests.

The recommended number of AIO VPs depends on how many disks your configuration supports. If KAIO is *not* implemented on your platform, Informix recommends that you allocate one AIO VP for each disk that contains database tables, and add an additional AIO VP for each chunk that the database server accesses frequently.

The machine-notes file for your version of the database server specifies whether the operating system supports KAIO. IF the operating system supports KAIO, the machine notes describe how to enable KAIO on your specific operating system.

If your operating system supports KAIO, the CPU VPs make unbuffered I/O requests directly to the operating system. In this case, configure only one AIO VP, plus two additional AIO VPs for every buffered operating-system file chunk.

The goal is to provide enough AIO VPs to keep the I/O request queues short; that is, the queues should have as few I/O requests in them as possible. When the I/O request queues remain consistently short, I/O requests are processed as fast as they occur. Use the **onstat -g ioq** command to monitor the length of the I/O queues for the AIO VPs. In the following sample output, **gfd 5** displays the read queue for a pipe, and **gfdwq 5** displays its write queue:

```
AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
 fifo   0      0      0        0        0        0        0
  adt   0      0      0        0        0        0        0
  msc   0      0      1       10        0        0        0
  aio   0      0      1    12939        8    12907        0
  pio   0      0      0        0        0        0        0
  lio   0      0      1        2        0        2        0
  gfd   3      0      4      496      117      379        0
  gfd   4      0      5   171228   171228        0        0
  gfd   5      0      0        0        0        0        0
  gfdwq 5      2      4       15        0       15        0
```

Monitor the **len** and **maxlen** fields of the I/O request queue. If **len** is usually greater than 10 or if **maxlen** is usually greater than 25, requests are not being serviced fast enough. If the disks or controllers are not already saturated, add more AIO VPs to improve request servicing.

Allocate enough AIO VPs to accommodate the peak number of I/O requests. Generally, it is not detrimental to allocate a few extra AIO VPs. You can use **onmode -p** to start additional AIO VPs while the database server is in on-line mode. You cannot drop AIO VPs in on-line mode.

NUMFIFOVPS

The NUMFIFOVPS configuration parameter specifies the number of FIF (first in, first out) VPs that the database server brings up initially on a coserver node. The database server uses two FIF virtual processors to process high-performance loads and unloads through named pipes. If you usually load and unload data through named pipes, add more FIF virtual processors. For more information on using named pipes to load and unload tables, refer to the *Administrator's Guide*.

To increase parallel execution of loads and unloads through named pipes, add FIF VPs. Use **onmode -p** to start additional FIF VPs while the database server is in on-line mode. You cannot drop FIF VPs in on-line mode.

PSORT_NPROCS

The database server almost always starts an appropriate number of sort threads for an PDQ queries. If CPU processing does not keep up with disk I/O for some queries, you might consider setting the **PSORT_NPROCS** environment variable for a client application that runs the queries. Generally, however, you should let the database server determine the number of sort threads required. The database server imposes an upper limit of 10 sort threads per query on each coserver.

For more information on parallel sorts and the **PSORT_NPROCS** environment variable**,** refer to "Parallel Sorts" on page 11-21.

NETTYPE

The NETTYPE configuration parameter configures poll threads for each network connection type that your instance of the database server supports. You must specify a separate NETTYPE parameter for each connection type if your database server instance supports connections over more than one network interface or protocol.

You usually include a separate NETTYPE parameter for each network connection type that is associated with a connection coserver. A coserver has a name of the following form:

```
dbservername.coserver-number
```

dbservername is the value that you specify in the DBSERVERNAME or DBSERVERALIASES configuration parameter.

coserver-number is the integer that you specify in each COSERVER configuration parameter.

The **sqlhosts** file associates connection types with the names of the connection coservers or the name of a dbserver group. For more information and an example of an **sqlhosts** file that specifies dbserver groups, refer to the *Administrator's Guide*.

The first NETTYPE entry for a given connection type in the ONCONFIG file applies to all coserver names that are associated with that type. Subsequent NETTYPE entries for that connection type are ignored. Even if connection types are not listed in the **sqlhosts** file, NETTYPE entries are required for connection types that are used for outgoing communication.

Each poll thread that a NETTYPE entry configures or adds dynamically runs in a separate VP. The two VP classes in which a poll thread can run are NET and CPU. Do not specify more poll threads than you need to support user connections. A poll thread run by a NET VP can handle at least 100 connections. Poll threads run by CPU VPs are faster than poll threads run by NET VPs, but they can handle only about 50 connections.

For best performance, Informix recommends that you assign only one poll thread to the CPU VP class with a NETTYPE entry and that you assign all additional poll threads to NET VPs.The maximum number of poll threads that you assign to any one connection type must not exceed NUMCPUVPS.

For best performance specify between 50 and 200 connections per poll thread, although in some environments a poll thread might be able to support as many as 400 connections.

Each NETTYPE entry configures the number of poll threads for a specific connection type, the number of connections per poll thread, and the virtual-processor class in which those poll threads run. The fields are separated by commas. No white space can exist within or between these fields:

```
NETTYPE connection_type,poll_threads,c_per_t,vp_class
```

connection_type identifies the protocol-interface combination to which the poll threads are assigned. You usually set this field to match the *connection_type* field of a coserver-name entry in the **sqlhosts** file.

poll_threads is the number of poll threads assigned to the connection type. Set this value to no more than NUMCPUVPS for any connection type. One poll thread can usually handle communications for up to 100 users.

c_per_t is the number of connections per poll thread. Use the following formula to calculate this number:

```
c_per_t = connections / poll_threads
```

connections is the maximum number of connections that you expect the indicated connection type to support. For shared-memory connections (**ipcshm**), double the number of connections for best performance.

vp_class is the class of VP that can run the poll threads. Specify CPU if you have a single poll thread that runs on a CPU VP. For best performance, specify NET if you require more than one poll thread. The default value for this field depends on the following conditions:

- If the connection type is associated with the coserver name that is listed in the DBSERVERNAME parameter, and no previous NETTYPE parameter specifies CPU explicitly, the default VP class is CPU. If the CPU class is already taken, the default is NET.
- If the connection type is associated with a coserver name that the DBSERVERALIASES parameter provides, the default VP class is NET.

If *c_per_t* exceeds `350` and the number of poll threads for the current connection type is less than NUMCPUVPS, you can improve performance by specifying the NET CPU class, adding poll threads (do not exceed NUMCPUVPS), and recalculating *c_per_t*. The default value for *c_per_t* is `50`.

Important: *Each* ***ipcshm*** *connection requires a semaphore. Some operating systems require that you configure a maximum number of semaphores that can be requested by all software packages that run on the computer. For best performance, double the number of actual* ***ipcshm*** *connections when you allocate semaphores for shared-memory communications. Refer to* *"UNIX Semaphore Parameters" on page 3-3*.

If your computer is a uniprocessor and your database server instance is configured for only one connection type, you can omit the NETTYPE parameter. The database server uses the information provided in the **sqlhosts** file to establish client / server connections.

If your computer is a uniprocessor and your database server instance is configured for more than one connection type, include a separate NETTYPE entry for each connection type. If the number of connections of any one type significantly exceeds 300, assign two or more poll threads, up to a maximum of NUMCPUVPS, and specify the NET VP class, as the following example shows:

```
NETTYPE ipcshm,1,200,CPU
NETTYPE tlitcp,2,200,NET # supports 400 connections
```

If your computer is a multiprocessor, your database server instance is configured for only one connection type, and the number of connections does not exceed 350, you can use NETTYPE to specify a single poll thread on either the CPU or the NET VP class. If the number of connections exceeds 350, set the VP class to NET, increase the number of poll threads, and recalculate *c_per_t*.

Virtual Processors and CPU Use

You can add virtual processors (VPs) to increase parallel execution. While the database server is on-line, it allows you to start and stop VPs that belong to certain classes. While the database server is on-line, you can use **onmode -p** to start additional VPs for the following classes: CPU, AIO, FIF, PIO, LIO, SHM, TLI, and SOC.

Whenever you add a network connection of the SOC or TLI class, you also add a poll thread. Every poll thread runs in a separate VP, which can be either a CPU VP or a network VP of the appropriate network type. Adding more VPs can increase the load on CPU resources, so if the NETTYPE value specifies that an available CPU VP can handle the poll thread, the database server assigns the poll thread to that CPU VP. If all the CPU VPs have poll threads assigned to them, the database server adds a second network VP to handle the poll thread.

Chapter

4

Effect of Configuration on Memory Use

In This Chapter

This chapter discusses how the combination of operating system and database server configuration parameters can affect memory use. It describes the parameters that most directly affect memory use and explains how to set them. It also suggests settings or considerations for different work loads.

When you allocate shared memory for the database server, consider the amount of physical memory that is available on each coserver host and the amount of memory that other applications require.

As a general rule, if you increase shared memory for the database server, you improve its performance.

Allocating Shared Memory for the Database Server

Shared memory is managed locally on each coserver. However, on the assumption that all coservers have the same amount of physical memory and to balance resources for parallel processing across coservers, the shared memory configuration parameters are set for all coservers in the global section of the ONCONFIG file.

Make sure that the shared-memory setting is adequate for the projected work load on each coserver. Configuring insufficient shared memory can adversely affect performance.

When the operating system allocates a block of shared memory, that block is called a *segment*. When the database server attaches all or part of a shared-memory segment, it is called a *portion*.

The database server uses the following shared-memory portions, each of which makes a separate contribution to the total amount of shared memory that the database server requires:

- Resident portion
- Virtual portion
- Message portion

The size of the resident and message portions of shared memory does not change after the database server is brought on line. You must allocate sufficient operating-system shared memory for these portions before you bring the database server into on-line mode. To reconfigure shared memory, you usually must reboot the operating system.

Although the size of the virtual portion of shared memory for the database server grows dynamically, you must still include an adequate initial amount for this portion in your allocation of operating-system shared memory.

The following sections provide guidelines for estimating the size of each shared-memory portion for the database server so that you can allocate adequate space. The amount of space required is the total that all three portions need initially.

For detailed information about shared memory, see the *Administrator's Guide*.

Resident Portion

The resident portion includes the coserver shared memory that records the state of the coserver, including buffers, locks, log files, and the locations of dbspaces, chunks, and tblspaces. Use the settings of the following global configuration parameters to estimate the size of this portion for each coserver:

- BUFFERS
- LOCKS
- LOGBUFF
- PHYSBUFF

In addition to these parameters, which affect the size of the resident portion of shared memory, the RESIDENT parameter can affect memory use. When RESIDENT is set to 1 in the ONCONFIG file of a computer that supports forced residency, the resident portion is never paged out. The machine-notes file for the database server specifies whether your operating system supports forced residency.

To estimate the size, in kilobytes, of shared memory required for the resident portion, follow the steps listed below. The resulting estimate slightly exceeds the actual memory used for the resident portion.

To estimate the size of the resident portion

1. Use the following formula to estimate the size of the data buffer:

   ```
   buffer_value = (BUFFERS * pagesize) + (BUFFERS * 254)
   ```

   In the formula, *pagesize* is the shared-memory page size. If you have specified a page size in the PAGESIZE configuration parameter, use this setting.

2. Use these formulas to calculate the following values:

   ```
   locks_value = LOCKS * 44
   logbuff_value = LOGBUFF * 1024 * 3
   physbuff_value = PHYSBUFF * 1024 * 2
   ```

3. Use the following formula to calculate the estimated size of the resident portion in kilobytes:

   ```
   rsegsize = (buffer_value + locks_value + logbuff_value
              + physbuff_value + 51,200) / 1024
   ```

For information about setting the BUFFERS, LOCKS, LOGBUFF, and PHYSBUFF configuration parameters, see "Configuration Parameters That Affect Memory Use" on page 4-10.

Virtual Portion

The virtual portion of shared memory for the database server includes the following components:

- Large private buffers, which are used for large read and write operations
- Query operations, such as sorts, hash joins, light scans, and groups
- Active thread-control blocks, stacks, and heaps

- User-session data
- Caches for data-dictionary information and SPL routines
- A global pool for network-interface message buffers and other information

The SHMVIRTSIZE configuration parameter in the database server configuration file specifies the initial size of the virtual portion. When the database server needs additional space in the virtual portion, it adds shared memory in increments as specified by the SHMADD configuration parameter, until it reaches the limit on the total shared memory that is specified by the SHMTOTAL parameter and allocated to the database server.

The optimal size of the virtual portion depends primarily on the types of applications and queries that run. Depending on your applications, an initial estimate for the virtual portion might be as low as 100 kilobytes per user, or as high as 500 kilobytes per user, plus an additional 4096 kilobytes (4 megabytes) for data distributions created by UPDATE STATISTICS, as described in "Creating Data-Distribution Statistics" on page 13-9.

Decision-support queries use large amounts of the virtual portion of shared memory to perform joins and sort operations.

In addition, if the communication interface between nodes in your system requires configurable buffers, you also need to consider the amount of space that these message buffers take up in the virtual portion of memory. For more details on these configurable buffers, refer to your machine-notes file.

For DSS applications, try to maximize the size of the virtual portion of shared memory relative to the resident portion. A common practice is to estimate the initial size of the virtual portion of shared memory as follows:

```
shmvirtsize = p_mem - os_mem - rsdnt_mem - (128K * users)
              - other_mem
```

The variables in the formula are defined as follows.

Variable	Description
p_mem	Total physical memory available on host
os_mem	Size of operating system, including buffer cache

(1 of 2)

Variable	Description
resdnt_mem	Size of resident shared memory for the database server
users	Number of expected users (connections) specified by the third argument of the NETTYPE configuration parameter
other_mem	Size of memory used for applications other than the database server applications

(2 of 2)

For more information on how to estimate the amount of the virtual portion of shared memory that the database server might need for sorting, refer to "Parallel Sorts" on page 11-21.

Tip: *When the database server is running with a stable work load, you can use* ***onstat -g mem*** *to obtain a precise value for the actual size of the virtual portion. You can then use the value for shared memory that this command reports to reconfigure SHMVIRTSIZE.*

Setting SHMVIRTSIZE to an appropriate size improves performance by reducing the CPU overhead that occurs when additional memory must be allocated dynamically for the virtual portion.

Message Portion

The message portion contains the message buffers that the shared-memory communication interface uses. The amount of space required for these buffers depends on the number of user connections that use a given network interface. If a particular interface is not used, you do not need to include space for it when you allocate shared memory in the operating system. You can use the following formula to estimate the size of the message portion in kilobytes:

```
msegsize = (10,531 * ipcshm_conn + 50,000)/1024
```

ipcshm_conn is the number of connections that can be made with the shared-memory interface, as determined by the NETTYPE parameter for the **ipcshm** protocol.

Configuring Shared Memory

After you calculated the shared memory required for the database server, perform the following steps to configure the shared-memory segments that your database server configuration needs. For information on how to set parameters related to shared memory, refer to the configuration instructions for your operating system.

To configure shared-memory segments

1. If your operating system does not have a limit for shared-memory segment size, take the following actions:
   a. Set the operating-system configuration parameter for maximum segment size, typically SHMMAX or SHMSIZE, to the total size that your database server configuration requires. This size includes the amount of memory that is required to initialize your database server instance and the amount of shared memory that you allocate for dynamic growth of the virtual portion.
   b. Set the operating-system configuration parameter for the maximum number of segments, typically SHMMNI, to at least 1 per instance of the database server.
2. If your operating system has a segment-size limit for shared memory, take the following actions:
   a. Set the operating-system configuration parameter for the maximum segment size, typically SHMMAX or SHMSIZE, to the largest value that your system allows.
   b. Use the following formula to calculate the number of segments for your instance of the database server. If there is a remainder, round up to the nearest integer.
      `SHMMNI = total_shmem_size / SHMMAX`

   In the formula, *total_shmem_size* is the total amount of shared memory that you allocate for database server use.

3. Set the operating-system configuration parameter for the maximum number of segments (typically SHMMNI) to a value that, when multiplied by SHMMAX or SHMSIZE, yields the total amount of shared memory for the database server, as estimated by using the formulas in "Allocating Shared Memory for the Database Server" on page 4-3.

   If nodes in your database server system are dedicated database server coservers, the total for each coserver can be up to 90 percent of the size of virtual memory (physical memory plus swap space).

4. If your operating system uses the SHMSEG configuration parameter to specify the maximum number of shared-memory segments that a process can attach, set this parameter to a value that is equal to or greater than the largest number of segments that you allocate for any instance of the database server.

For additional tips on configuring shared memory in the operating system, refer to the machine notes file for the database server.

Freeing Shared Memory

The database server does not automatically free the shared-memory segments that it adds during its operations. Once memory has been allocated to the database server virtual portion, the memory is not available to other processes that are running on the host computer. When the database server runs a large decision-support query, it might acquire a large amount of shared memory. After the query is complete, that shared memory is no longer required. However, the shared memory that was allocated to the query remains assigned to the virtual portion of shared memory for the database server even though it might no longer be needed.

The **onmode -F** command locates and returns unused 8-kilobyte blocks of shared memory that the database server still holds. Although this command runs only briefly (one or two seconds), **onmode -F** dramatically inhibits user activity while it runs. Systems with multiple CPUs and CPU VPs usually experience less performance degradation while this utility runs.

Informix recommends that you run **onmode -F** on each coserver during slack periods with an operating-system or other scheduling facility, such as **cron** on UNIX database servers. In addition, consider running this utility after you perform any task that substantially increases the size of database server shared memory, such as large decision-support queries, index builds, sorts, or backup operations. For additional information on the **onmode** utility, refer to the *Administrator's Reference*.

Configuration Parameters That Affect Memory Use

The following configuration parameters have a significant effect on memory use:

- BUFFERS
- DS_ADM_POLICY
- DS_MAX_QUERIES
- DS_TOTAL_MEMORY
- LOCKS
- LOGBUFF
- MAX_PDQPRIORITY
- PAGESIZE
- PDQPRIORITY
- PHYSBUFF
- SHMADD
- SHMBASE
- SHMTOTAL
- SHMVIRTSIZE
- STACKSIZE
- RESIDENT

The following sections describe the performance considerations associated with these parameters. For information about the database server ONCONFIG file, refer to the *Administrator's Guide*. For additional information about each configuration parameter, refer to the *Administrator's Reference*.

The guidelines for setting the following memory configuration parameters might be different for different kinds of application programs.

Parameter	OLTP	DSS
BUFFERS	Set to 20 to 25 percent of the number of megabytes in physical memory. If the levels of paging activity rises, reduce the value of BUFFERS.	Set a small buffer value and increase the DS_TOTAL_MEMORY value for queries and sorts. For operations such as index builds that read data through the buffer pool, configure a larger number of buffers.
DS_TOTAL_MEMORY	Set to 20 to 50 percent of the value of SHM_TOTAL, in kilobytes.	Set to 50 to 90 percent of SHM_TOTAL.
LOGBUFF	If you are using unbuffered or ANSI logging, use the **pages/io** value in the logical-log section of the **onstat -l** output for the LOGBUFF value. If you are using buffered logging, keep the **pages/io** value low. The recommended LOGBUFF value is 16 to 32 kilobytes or 64 kilobytes for heavy workloads.	Because database or table logging is usually turned off for DSS applications, set LOGBUFF to 32 kilobytes.
PHYSBUFF	If applications are using physical logging, check the **pages/io** value in the physical-log section of the **onstat -l** output to make sure the I/O activity is not too high. Set PHYSBUFF to a value that is divisible by the page size. The recommended PHYSBUFF value is 16 pages.	Because most DSS applications do not physically log, set PHYSBUFF to 32 kilobytes.
PAGESIZE	2048 kilobytes	8192 kilobytes

For information about how the Resource Grant Manager (RGM) uses the settings of these configuration parameters, see "Scheduling Queries" on page 12-7.

BUFFERS

The BUFFERS configuration parameter specifies the number of data buffers available to the database server. These buffers reside in the resident portion of shared memory and are used to cache database data pages. Because the BUFFERS parameter can be set to 2^{31-1}, it has no theoretical limit.

For OLTP applications, if more buffers are available, a required data page is more likely to be in memory as the result of a previous request. For example, if client applications access 15 percent of the data 90 percent of the time, set the BUFFERS parameter large enough to hold that 15 percent. Such a setting improves database I/O and transaction throughput. However, allocating too many buffers can affect the memory-management system and lead to excess paging activity.

To optimize some operations such as index builds that read data through the buffer pool, configure a larger number of buffers.

Informix suggests that you set BUFFERS to provide a buffer-space value between 20 and 25 percent of the number of megabytes in physical memory. Informix recommends that you calculate all other shared-memory parameters after you set buffer space (BUFFERS multiplied by the system page size) to 20 percent of physical memory. Then, after you specify all shared memory parameters, increase the size of BUFFERS to the full 25 percent if you have enough shared memory. If you set the page size in the PAGESIZE configuration parameter, described in "PAGESIZE" on page 4-21, make sure that you use the specified page size in your calculation.

Important: *The sizes of buffers for TCP/IP connections, which are specified in the* ***sqlhosts*** *file, affect memory and CPU use. Adjusting the size of these buffers to accommodate a typical request can improve CPU use by eliminating the need to break requests into multiple messages. However, because the database server dynamically allocates buffers of the specified sizes for active connections, buffers can consume large amounts of memory unnecessarily if you do not size them carefully.*

Tuning BUFFERS

Consider the example of a system with a page size of 4 kilobytes and 100 megabytes of physical memory. You can first set BUFFERS between `10,000` and `12,500` (40 to 50 megabytes). Then use **onstat -p** to monitor the read-cache rate, which is the percentage of database pages that are already present in a shared-memory buffer when a query requests them. A high read-cache rate is desirable for good performance because the database server copies pages into memory from disk if they are not already present.

If the read-cache rate is low, you can repeatedly increase BUFFERS and restart the database server. As you increase the value of BUFFERS, you reach a point at which increasing the value no longer produces significant gains in the read-cache rate, or you reach the upper limit of your operating-system shared-memory allocation. Use an operating-system utility that monitors memory, such as **vmstat** or **sar**, to monitor the level of page scans and paging-out activity. If these levels suddenly rise, or rise to unacceptable levels during peak database activity, reduce the value of BUFFERS.

For SMP coserver systems, such as those that have four CPUs on each node and as few as eight nodes, UPDATE STATISTICS can execute much faster if you set BUFFERS as high as `25,000` (100 megabytes) on each node. For uniprocessor coserver systems, you can set BUFFERS lower if there are more coserver nodes to do the work.

DS_ADM_POLICY

The DS_ADM_POLICY configuration parameter specifies how the RGM should schedule queries:

- If DS_ADM_POLICY is set to STRICT, the RGM processes queries in the order determined by the SET SCHEDULE LEVEL value that is specified in the query and the order in which queries are submitted. The RGM processes the oldest query with the highest scheduling level before other queries, which means that a more recent query with a lower scheduling level might never run or might not run for a long time.
- If DS_ADM_POLICY is set to FAIR, the RGM takes scheduling level, PDQ priority, and wait time into account when it decides which query to process. In general, the query with the highest scheduling level runs first, but a query with a lower scheduling level runs if it has been waiting a long time.

For more information about how the RGM uses DS_ADM_POLICY to manage queries, refer to "Using the Admission Policy" on page 12-8.

DS_MAX_QUERIES

The DS_MAX_QUERIES configuration parameter specifies the maximum number of memory-consuming queries that can run at any one time. A memory-consuming query is usually a DSS query that performs complex tasks that might include scans of entire tables, manipulation of large amounts of data, multiple joins, and the creation of temporary tables. The RGM manages memory-consuming queries.

PDQPRIORITY specifies the amount of memory that a query requests. Queries with a low PDQPRIORITY setting request proportionally smaller amounts of memory, so more of those queries can run simultaneously.

You can use the DS_MAX_QUERIES parameter to limit the performance impact of memory-consuming queries. The RGM enforces this limit. For more information, refer to "Limiting the Maximum Number of Queries" on page 12-17.

The RGM reserves memory for a query based on the following formula:

```
memory_reserved =(DS_TOTAL_MEMORY * (PDQPRIORITY / 100 *
            (MAX_PDQPRIORITY / 100)
```

To allow for a larger number of simultaneous queries with less memory each, increase DS_MAX_QUERIES. The maximum memory allowed is 8 megabytes.

For more information on how memory is allocated to queries, refer to "DS_TOTAL_MEMORY" on page 4-15.

DS_TOTAL_MEMORY

The DS_TOTAL_MEMORY configuration parameter places a ceiling on the amount of shared memory that queries can obtain on each coserver. You can use this parameter to control the effect that large, memory-intensive queries have on other queries and transactions. The higher you set this parameter, the more memory a memory-intensive query can use, and the less memory is available to process other queries and transactions. The maximum you can specify is 2 gigabytes, or half of the maximum of 4 gigabytes for SHMVIRTSIZE.

To allow for a larger number of simultaneous queries that require a relatively small amount of memory, increase DS_MAX_QUERIES. For more information, refer to "DS_MAX_QUERIES" on page 4-14.

The RGM uses the value that you specify in DS_TOTAL_MEMORY to allocate memory to queries. For a figure that shows the relation of the memory-configuration parameters for DSS queries and provides more information about how the RGM uses the DS_TOTAL_MEMORY setting, refer to "How the RGM Grants Memory" on page 12-5.

Initial Value Estimate for DS_TOTAL_MEMORY

Use the following formula as a starting point for estimating the amount of shared memory to allocate to decision-support queries:

```
DS_TOTAL_MEMORY = p_mem - nondecision_support_memory
```

The value of *p_mem* is the total physical memory available on the computer.

Use the following formula to estimate the amount of memory required for other queries and other applications:

```
nondecision_support_memory = os_mem - rsdnt_mem -
               (128 kilobytes * users) - other_mem
```

Variable	Description
os_mem	Size of operating system, including buffer cache
rsdnt_mem	Size of Informix resident shared memory
users	Number of expected users (connections) that the third argument of the NETTYPE configuration parameter specifies
other_mem	Size of memory used for other (non-Informix) applications

For OLTP applications, set DS_TOTAL_MEMORY to between 20 and 50 percent of the value of SHMTOTAL, in kilobytes. For DSS applications, set the value of DS_TOTAL_MEMORY to between 50 and 80 percent of SHMTOTAL. If your database server system is used exclusively for DSS queries, set this parameter to 90 percent of SHMTOTAL.

Algorithm to Determine DS_TOTAL_MEMORY

If you do not set DS_TOTAL_MEMORY, or if you set it to an inappropriate value, the database server derives an appropriate value. If the database server changes the value that you assigned to DS_TOTAL_MEMORY, it sends the following message to the location specified by the MSGPATH configuration parameter:

```
DS_TOTAL_MEMORY recalculated and changed from old_value kilobytes
            to new_value kilobytes
```

If this message appears, use the algorithm that the following sections describe to investigate the values that the database server considers inappropriate. You can then make adjustments based on your investigation.

Derive a Minimum for Decision-Support Memory

In the first part of the algorithm, the database server establishes a minimum for decision-support memory. The database server uses the following formula to set the minimum amount of decision-support memory:

```
min_ds_total_memory = NUMCPUVPS * 4 * 128 kilobytes
```

Derive a Working Value for Decision-Support Memory

In the second part of the algorithm, the database server establishes a working value for the amount of decision-support memory. The database server verifies this amount in the third and final part of the algorithm, as follows:

- When DS_TOTAL_MEMORY is set, the database server first checks the SHMTOTAL setting. If SHMTOTAL is set, the database server uses the following formula to calculate the amount of decision-support memory:

  ```
  IF DS_TOTAL_MEMORY <= SHMTOTAL -
  nondecision_support_memory THEN
     decision_support_memory = DS_TOTAL_MEMORY
  ELSE
      decision_support_memory = SHMTOTAL -
                    nondecision_support_memory
  ```

  This algorithm prevents you from setting DS_TOTAL_MEMORY to values that the database server cannot allocate to decision-support memory.

  For information about estimating an appropriate amount of memory for OLTP applications, see "Initial Value Estimate for DS_TOTAL_MEMORY" on page 4-15.

  If SHMTOTAL is not set, the database server sets decision-support memory to the value that you specified in DS_TOTAL_MEMORY.

- When DS_TOTAL_MEMORY is not set, the database server proceeds differently. First, the database server checks the setting of SHMTOTAL. If SHMTOTAL is set, the database server uses the following formula to calculate the amount of decision-support memory:

  ```
  decision_support_memory = SHMTOTAL -
                    nondecision_support_memory
  ```

  If SHMTOTAL is not set, the database server sets decision-support memory as the following example shows:

  ```
  decision_support_memory = min_ds_total_memory
  ```

  For information about setting the variable **min_ds_total_memory**, see "Derive a Minimum for Decision-Support Memory" on page 4-17.

Check Derived Value for Decision-Support Memory

The final part of the algorithm verifies that the amount of shared memory is greater than **min_ds_total_memory** and less than the maximum possible amount of memory for the computer. When the database server finds that the derived value for decision-support memory is less than **min_ds_total_memory**, it sets decision-support memory equal to **min_ds_total_memory**.

When the database server finds that the derived value for decision-support memory is greater than the maximum possible amount of memory for the computer, it sets decision-support memory equal to the maximum possible memory.

Inform User When Derived Value Is Different from User Value

At any point during the processing of this algorithm, if the database server changes the value that you set for DS_TOTAL_MEMORY, it sends a message to your console in the following format:

```
DS_TOTAL_MEMORY recalculated and changed from old_value kilobytes
            to new_value kilobytes
```

In the message, *old_value* represents the value that you assigned to DS_TOTAL_MEMORY in your configuration file, and *new_value* represents the value that the database server derived.

LOCKS

The LOCKS parameter sets the initial number of locks that can be used at any one time. Each lock requires 44 bytes in the resident segment. You must provide for this amount of memory when you configure shared memory.

If the database server needs more locks, 100,000 additional locks are added and a message is written to the event log, indicating that this has happened. Locks can be increased as many as 32 times before the database server reports that it is out of locks. If you frequently see the dynamic-lock- allocation message, increase the value of the LOCKS parameter. The maximum you can specify is 16,000,000 locks.

Set LOCKS to the number of locks that queries usually need, multiplied by the number of concurrent users. To estimate the number of locks that a query needs, use the guidelines in the following table.

Locks per SQL statement	Isolation Level	Table	Row	Key	Simple Large Object
SELECT	Dirty Read	0	0	0	0
	Committed Read	1	0	0	0
	Cursor Stability	1 or value specified in ISOLATION_LOCKS	1 or value specified in ISOLATION_LOCKS	0	0
	Indexed Repeatable Read	1	Number of rows satisfying conditions	Number of rows satisfying conditions	0
	Sequential Repeatable Read	1	0	0	0

(1 of 2)

Locks per SQL statement	Isolation Level	Table	Row	Key	Simple Large Object
INSERT	For locks, relevant only for SELECT statements.	1	1	Number of indexes	Number of pages in simple large objects
DELETE	For locks, relevant only for SELECT statements.	1	1	Number of indexes	Number of pages in simple large objects
UPDATE	For locks, relevant only for SELECT statements.	1	1	2 per changed key value	Number of pages in old plus new simple large objects

(2 of 2)

Important: *During a DROP DATABASE operation, the database server acquires and holds a lock on each table in the database until the entire DROP operation is complete. If the value of LOCKS is large enough to accommodate the largest number of tables in a database, dynamic lock allocation is not required.*

LOGBUFF

The LOGBUFF parameter specifies the amount of shared memory that is reserved for each of the three buffers that hold the logical-log records until they are flushed to the logical-log file on disk. The size of a buffer determines how often it fills and therefore how often it must be flushed to the logical-log file on disk.

Important: *The value of LOGBUFF depends in part on the page size that you specify for your database server. If you used the PAGESIZE configuration parameter, described in* *"PAGESIZE" on page 4-21**, to set the page size, make sure that you use the specified page size in your estimate of an appropriate LOGBUFF value.*

MAX_PDQPRIORITY

The MAX_PDQPRIORITY configuration parameter specifies the percentage of memory that a single query can use and thus limits the effect of large CPU- and memory-intensive queries on transaction throughput and makes it possible for more queries to run during the same time period.

Users who issue queries can set the **PDQPRIORITY** environment variable or use the SET PDQPRIORITY statement in SQL to override the PDQPRIORITY configuration parameter setting. The setting of the MAX_PDQPRIORITY configuration parameter limits the amount of memory that is actually granted.

Setting MAX_PDQPRIORITY helps to balance a system that runs both OLTP clients and DSS queries. To allocate more resources to OLTP processing, reduce the value of MAX_PDQPRIORITY. To allocate more resources to decision-support processing, increase the value of MAX_PDQPRIORITY. For more information on how to control PDQ resource use, refer to "Managing Resources for DSS and OLTP Applications" on page 12-11.

For more information about PDQ memory and an illustration of the way in which the PDQ configuration parameters are related, refer to "How the RGM Grants Memory" on page 12-5. For more information about the environment variable and the SQL statement, refer to the *Informix Guide to SQL: Reference* and the *Informix Guide to SQL: Syntax*, respectively.

PAGESIZE

Use the PAGESIZE configuration parameter to specify the page size that the the database server uses. You can specify one of three sizes: 2048 bytes, 4096 bytes, or 8192 bytes. If you do not set the PAGESIZE parameter, the database server uses a default size of 4096 bytes.

A page size as small as 2048 bytes might improve performance on a database server that is used for applications that access tables randomly, such as OLTP applications. A larger page size might improve performance on a database server that is used for applications that access tables sequentially, such as certain DSS applications.

If you set the PAGESIZE configuration parameter, use the setting in the formulas for which the page size is required to estimate other configuration parameter settings. Some of the affected parameters are PHYSBUFF, LOGBUFF, and BUFFERS.

For more information about the PAGESIZE parameter, refer to the *Administrator's Reference*.

PDQPRIORITY

THE PDQPRIORITY configuration parameter provides a minimum and optional maximum value for the percentage of shared memory that an individual query can use. The Resource Grant Manager (RGM) does not run a memory-consuming query until the minimum requested amount of memory is available.

If you specify only a single percentage for PDQPRIORIY, the RGM can run a query only when the specified amount of memory is available. If you set a range of resources from the minimum required for a query to an optimal maximum, the RGM can run a query when available memory falls in the specified range.

You can set PDQPRIORITY in the following ways:

- With the PDQPRIORITY configuration parameter

  The syntax is as follows:

  ```
  PDQPRIORITY lowval [,highval]
  ```

  Set the PDQPRIORITY configuration parameter to provide a database-server-wide default. To modify this default, users can set the environment variable or use the SQL statement. The setting of MAX_PDQPRIORITY scales the amount of memory actually granted.

- With the **PDQPRIORITY** environment variable

  The UNIX syntax for the **PDQPRIORITY** environment variable is as follows:

  ```
  export PDQPRIORITY="lowval [,highval]"
  ```

  When **PDQPRIORITY** is set in the environment of a client application, it specifies the percentage of shared memory that can be allocated to any query that client starts.

- From a client program that uses the SET PDQPRIORITY statement

  The syntax of the SET PDQPRIORITY statement is as follows:

  ```
  SET PDQPRIORITY LOW lowval HIGH highval;
  ```

  If a client application uses the SET PDQPRIORITY statement in SQL to set a value for PDQPRIORITY, that value overrides the configuration parameter and environment variable settings. The setting of MAX_PDQPRIORITY limits the actual percentage of resources granted.

If PDQPRIORITY is not set or is set to `0`, each SQL operator instance required by a query is granted 128 kilobytes of memory. For example, on a single coserver system with one CPU VP, a query with two hash joins is granted `2 * 128`, or 256 kilobytes of memory. On a single coserver system with two CPU VPs, each hash join SQL operator has two instances, so the query is granted `2 * 2 * 128`, or 512 kilobytes of memory.

For more information on limiting resources that can be allocated to queries, see "DS_TOTAL_MEMORY" on page 4-15 and "MAX_PDQPRIORITY" on page 4-21. For information about how the RGM uses PDQPRIORITY and related parameters, see Chapter 12, "Resource Grant Manager."

PHYSBUFF

The PHYSBUFF configuration parameter specifies the amount of shared memory that is reserved for each of the two buffers that serve as temporary storage space for data pages that are about to be modified. The size of a buffer determines how often it fills and therefore how often it must be flushed to the physical log on disk.

The value of PHYSBUFF depends in part on the page size specified for your database server by the PAGESIZE configuration parameter, described in "PAGESIZE" on page 4-21.

RESIDENT

The RESIDENT configuration parameter specifies whether shared-memory residency is enforced for the resident portion of database server shared memory. You can use this parameter only on computers that support forced residency.

The resident portion in the database server contains the least-recently used (LRU) queues for database read and write activity. Performance improves when these buffers remain in physical memory. Informix recommends that you set the RESIDENT parameter to 1. If forced residency is not an option on your computer, the database server issues an error message and ignores this parameter.

You can turn residency on or off for the resident portion of shared memory in the following ways:

- Use the **onmode** utility to reverse the state of shared-memory residency while the database server is on-line.
- Change the RESIDENT parameter to turn shared-memory residency on or off the next time that you initialize database server shared memory.

SHMADD

The SHMADD CONFIGURATION parameter specifies the size of each increment of shared memory that the database server dynamically adds to the virtual portion on each coserver. The maximum allowable size of a memory increment is 4 gigabytes.

When the database server adds shared memory, it consumes CPU cycles. Consider the trade-offs when you determine the size for SHMADD:

- Large increments are generally preferable to reduce the number of times that the database server adds shared memory. However, less memory is available for other processes.
- If memory is heavily loaded, as indicated by a high scan or paging-out rate, smaller increments allow better sharing of memory resources among competing programs.

Specify the same size for the SHMADD parameter on all coservers. The database server tries to balance the workload dynamically across all coservers to use the resources on each coserver equally.

Informix suggests that you set SHMADD according to the size of physical memory, as the following table indicates.

Physical Memory Size on Each Coserver	SHMADD Value for Each Coserver
256 megabytes or less	8192 kilobytes (the default)
Between 257 and 512 megabytes	16,384 kilobytes
More than 512 megabytes	Up to 65,536 kilobytes

The size of segments that you add should match the size of segments allocated in the operating system. For more information about configuring shared-memory segments, refer to "Configuring Shared Memory" on page 4-8. Some operating systems place a lower limit on the size of a shared-memory segment; your setting for SHMADD should be more than this minimum.

To find out how many shared-memory segments the database server is currently using, use the **onstat -g seg** command on a specific coserver or use **xctl onstat -g seg** to display shared memory information for all coservers. The following example shows **onstat-g seg** output.

```
Segment Summary:
id       key         addr           size     ovhd      class    blkused blkfree
13406    1382029314  c0b74000        155648   616      M        6       32
13805    1382029313  c1777000        130662409512      R*       3160    30
(shared) 1382029313  c23ed000        2684477448804     V        10656   54883
Total:                 -             281669632                  13822   54945
```

Figure 4-1
onstat -g seg Output

In Figure 4-1, **size** displays the number of bytes in the segment, **blkused** displays the number of blocks in the segment in page units, and **blkfree** displays the number of free blocks in the segment page units. To calculate the number of bytes in a segment, multiply the number of blocks by the page size that you specified in the PAGESIZE configuration parameter. The default is 4 kilobytes.

SHMBASE

The SHMBASE parameter specifies the starting address for the database server shared memory. When you set this parameter according to the instructions in your machine notes, it does not affect performance.

SHMTOTAL

The SHMTOTAL configuration parameter sets an upper limit on the amount of shared memory that each coserver can use. If SHMTOTAL is set to `0` or left unassigned, the database server continues to attach additional memory as needed until no more shared memory is available on the system. For information about setting an appropriate value for SHMTOTAL on your system, refer to your machine notes.

If your operating system runs out of swap space and performs abnormally, set SHMTOTAL to a value that is a few megabytes less than the total swap space that is available on each coserver. If SHMTOTAL is set to `0`, memory might be exhausted on the database server, and the database server will hang.

If SHMTOTAL is set to 0 and you have an exceptionally heavy workload, configure swap space that is from one and a half to two times the size of physical memory.

SHMVIRTSIZE

The SHMVIRTSIZE configuration parameter specifies the size of the virtual portion of shared memory that is initially allocated to the database server. The virtual portion of shared memory holds session- and request-specific data as well as other information, as described in "Virtual Portion" on page 4-5. The maximum amount of a virtual shared-memory partition is 4 gigabytes.

Although the database server adds increments of shared memory to the virtual portion as needed to process large queries or handle peak loads, the necessity of allocating additional shared memory increases time for transaction processing. For this reason, Informix recommends that you set SHMVIRTSIZE large enough to cover most normal daily operating requirements. For an initial setting for database servers that run OLTP applications, Informix suggests that you use the larger of the following values:

- 8000 kilobytes
- *connections* * 350

  In the formula, *connections* is the number of connections for all network types that are specified in the **sqlhosts** file by one or more NETTYPE parameters. (The database server uses *connections* * 200 by default.)

When system use reaches a stable workload, you can reconfigure a new value for SHMVIRTSIZE. "Freeing Shared Memory" on page 4-9 explains how you can release shared-memory segments that are no longer in use after a peak workload period or large query.

STACKSIZE

The STACKSIZE configuration parameter specifies the initial stack size for each thread. The database server assigns this amount of space to each active thread. The database server allocates space for thread stacks from the virtual portion of shared memory.

To reduce the amount of shared memory that the database server adds dynamically, estimate the amount of stack space required for the average number of threads that your system runs. Then include that amount in the value you set for SHMVIRTSIZE. To estimate the amount of stack space that you require, use the following formula:

```
stacktotal = STACKSIZE * avg_no_of_threads
```

avg_no_of_threads is the average number of threads.

Monitor the number of active threads at regular intervals to determine this amount. Use **onstat -g sts** to check the stack use of threads. A general estimate is between 60 and 70 percent of the total number of connections (specified in the **sqlhosts** file or with NETTYPE parameters), depending on your workload.

Chapter

5

Effect of Configuration on I/O

In This Chapter

This chapter describes how your database server configuration affects I/O activity. It covers the following topics:

- Configuring chunks and dbspaces
- Managing critical data
- Managing temporary space
- Managing I/O for tables
- Managing background I/O activities

Chunk and Dbspace Configuration

All of the data in a database is stored on disk. How fast the database server can copy required data pages to and from disk is important in improving performance.

Disks are usually the slowest component in the I/O path for a transaction or query that runs entirely on one host computer. Network communication can also introduce delays in client/server applications, but these delays are usually outside of the control of the database server administrator.

Disks can become overused or saturated when pages are requested often. Saturation can also occur when you use a disk for multiple purposes, such as for both logging and active database tables, when disparate data resides on the same disk, or when table extents are interleaved.

The various functions that your application performs, as well as the consistency-control functions that the database server performs, determine the optimal disk, chunk, and dbspace layout for your database. Chapter 6, "Table Performance," discusses these factors. The more coserver nodes you make available to the database server, the easier it is to balance I/O across them.

This section outlines important issues for the initial configuration of your chunks and dbspaces. Consider the following issues when you decide how to lay out chunks and dbspaces on disks:

- Placement and mirroring of critical database server data
- Load balancing
- Reducing contention
- Ease of backup and restore

The number of chunks, dbslices, and dbspaces that you can create is determined by the CONFIGSIZE configuration parameter, and its overriding parameters, MAX_DBSLICES, MAX_CHUNKS, and MAX_DBSPACES. For information about these parameters, refer to the *Administrator's Reference*.

Associate Disk Partitions with Chunks

Informix recommends that you create chunks that occupy entire disk partitions. When a chunk coincides with a disk partition (or device), you can easily track disk-space use, and you avoid errors caused by miscalculated offsets.

On the other hand, if your parallel-processing platform has a high-speed interconnect, so that disk seek and transfer time is a more significant factor in query response performance, you might consider creating a chunk in the middle cylinders of the disk and putting your high-access tables in a dbspace that is associated with this chunk. In general, seek time is shorter for data in the middle cylinders of the disk because the read / write heads move a shorter distance.

For more information about fragmenting tables across coservers, see "Specifying Table Placement" on page 6-12.

Associate Dbspaces with Chunks

Informix recommends that you associate a single chunk with a dbspace, especially when you plan to use that dbspace for a table fragment. For more information on table placement and layout, refer to Chapter 6, "Table Performance."

When a disk that contains the system catalog for a particular database fails, the entire database is inaccessible until the system catalog is restored. For this reason, Informix recommends that you do not cluster the system catalog tables for all databases in a single dbspace but instead place the system catalog tables with the database tables that they describe. System catalog tables require about 1.5 megabytes of disk space.

To create the database system catalog tables in the table dbspace

1. Create a database in the dbspace in which the table is to reside.
2. Use the SQL statement DATABASE or CONNECT to make that database the current database.
3. Enter the CREATE TABLE statement to create the table.

Management of Critical Data

The disk or disks that contain the system reserved pages, the physical log, and the dbspaces that contain the logical-log files are critical to the operation of the database server. It cannot operate if any of these elements is unavailable. By default, the database server places all three critical elements in the root dbspace on each coserver.

The root dbspace for each coserver is specified in the ROOTDBSLICE configuration parameter. By default, the root dbslice is **rootdbs**. The root dbspaces on the coservers are **rootdbs.1**, **rootdbs.2**, and so on.

To decide on an appropriate placement strategy for critical database server data, you must balance the necessity of protecting data availability and allowing maximum logging performance.

To prevent the database server from placing temporary tables and sort files in critical dbspaces, use the default setting, NOTCRITICAL, for the DBSPACETEMP configuration parameter in the ONCONFIG file. Even better, use the **onutil** utility to create dbspaces or dbslices explicitly for temporary file use only and assign these spaces to the DBSPACETEMP configuration parameter. For details, see "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

Separate Disks for Critical Data

Placing the root dbspace, logical log, and physical log in separate dbspaces on separate disks has performance benefits. The disks that you use for each critical database server component should be on separate controllers. The benefits of this separation are as follows:

- Logging activity is isolated from database I/O and allows physical-log I/O requests to be serviced in parallel with logical-log I/O requests.
- Time needed to recover from a failure is reduced.

However, if a disk that contains critical data fails, the database server halts and requires complete restoration of all data from a level-0 backup unless the dbspaces that contain critical database server data are mirrored.

If you separate the logical and physical logs from the root dbspace, you need a relatively small root dbspace that contains only reserved pages, the database partition, and the **sysmaster** database. In many cases, 10,000 kilobytes is sufficient.

The database server configures different portions of critical data differently:

- To move the physical log files to a dedicated dbslice, create the dbslice and set the PHYSSLICE database server configuration parameter.
- To assign the logical-log files to a dedicated dbslice, create the dbslice and use the **onutil** CREATE LOGICAL LOGSLICE command.

For more information about relocating the logical and physical logs, refer to your *Administrator's Guide*.

"Configuration Parameters That Affect Critical Data" on page 5-9 describes the configuration parameters that affect each portion of critical data.

Mirroring for Critical Data

Mirroring the dbspaces that contain critical data ensures that the database server can continue to operate when a single disk fails. However, the mix of read and write I/O requests for a specific dbspace determines whether I/O performance suffers if the dbspace is mirrored. A noticeable performance advantage occurs when you mirror dbspaces that have a read-intensive use pattern, and a slight performance disadvantage occurs when you mirror write-intensive dbspaces.

When mirroring is in effect, two disks are available to handle read requests, and the database server can process a higher volume of those requests. However, each write request requires two physical write operations and is not complete until both physical operations are performed. The write operations are performed in parallel, but the request is not complete until the slower of the two disks performs the update. Thus, you experience a slight performance penalty when you mirror write-intensive dbspaces.

Mirroring the Root Dbspace

You can achieve a certain degree of fault tolerance with a minimum performance penalty if you mirror the root dbspace and restrict its contents to read-only or seldom-accessed tables. When you place tables that are updated often in other, nonmirrored dbspaces, use the database server backup and restore facilities to perform warm restores of those tables if a disk fails. When the root dbspace is mirrored, the database server remains on-line to service other transactions while the failed disk is being repaired.

When you mirror the root dbspace, always place the first chunk on a different device than that of the mirror. The value of the MIRRORPATH configuration parameter should be different from the value of ROOTPATH.

Mirroring the Logical Log

The logical log is write intensive. If the dbspace that contains the logical-log files is mirrored, you encounter the slight double-write performance penalty described in “Mirroring for Critical Data” on page 5-7. However, if you choose an appropriate log buffer size and logging mode, you can adjust the rate at which logging generates I/O requests to a certain extent.

With unbuffered and ANSI-compliant logging, the database server requests one flush of the log buffer to disk for every committed transaction and two when the dbspace is mirrored. Buffered logging generates far fewer I/O requests than unbuffered or ANSI-compliant logging. With buffered logging, the log buffer is written to disk only when it fills and all the transactions that it contains are completed. You can reduce the frequency of logical-log I/O even more if you increase the size of your logical-log buffers. However, with buffered logging you might lose transactions in partially filled buffers if the system fails.

Although database consistency is guaranteed under buffered logging, specific transactions are not guaranteed against a fault. The larger the logical-log buffers, the more transactions you might need to reenter when service is restored after a fault.

You cannot specify an alternative dbspace for logical-log files in your initial database server configuration as you can for the physical log. To add logical-log files to a different dbspace and then drop the logical-log files in the root dbspace, follow these two steps:

1. Use the **onutil** CREATE LOGICAL LOG command to add logical-log files to an alternative dbspace.
2. Use the **onutil** DROP LOGICAL LOG command to drop logical-log files from the root dbspace.

For more information about **onutil**, refer to the *Administrator's Reference*.

Mirroring the Physical Log

The physical log is write intensive, with activity occurring at checkpoints and when buffered data pages are flushed to the disk. I/O to the physical log also occurs when a page-cleaner thread is activated. If the dbspace that contains the physical log is mirrored, you encounter the slight double-write performance penalty noted in "Mirroring for Critical Data" on page 5-7. To keep I/O to the physical log at a minimum, adjust the checkpoint interval and the LRU minimum and maximum thresholds. For more information, see "CKPINTVL" on page 5-23 and "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-28.

Configuration Parameters That Affect Critical Data

Use the following configuration parameters to configure the root dbspace and its mirror components:

- MIRROR
- MIRROROFFSET
- MIRRORPATH
- ROOTNAME
- ROOTOFFSET
- ROOTPATH
- ROOTSIZE
- ROOTSLICE

These parameters determine the location and size of the initial chunk of the root dbspace and configure mirroring, if any, for that chunk. (If the initial chunk is mirrored, all other chunks in the root dbspace must also be mirrored). These parameters have no other major impact on performance.

The following configuration parameters affect the logical logs:

- LOGBUFF
- LOGFILES
- LOGSIZE
- LOGSMAX

LOGBUFF determines the size of the three logical-log buffers that are in shared memory. For more information on LOGBUFF, refer to "LOGBUFF" on page 4-20. LOGFILES specifies the number of logical log files on each coserver when the database server is initialized. LOGSIZE determines the size of each logical log fix, and LOGSMAX specifies the maximum number of logical-log files. LOGSMAX must always be larger than LOGFILES.

The following configuration parameters determine the location and size of the physical log:

- PHYSDBS
- PHYSFILE

Dbspaces for Temporary Tables and Sort Files

Applications that use temporary tables or large sort operations require a large amount of temporary space.

You can improve performance with the use of *temporary dbspaces* or *temporary dbslices* that you create exclusively to store temporary tables and sort files. Specify these specially created dbspaces and dbslices as arguments to the DBSPACETEMP configuration parameter and the **DBSPACETEMP** environment variable to ensure that the database server uses these spaces for temporary tables and sort files.

The database server uses the value of the DBSPACTEMP configuration parameter or environment variable for the temporary table and sort files in the following circumstances:

- When you create an explicit temporary table with the INTO SCRATCH or INTO TEMP option of the SELECT statement or with the SCRATCH TABLE or TEMP TABLE clause of the CREATE TABLE statement, unless you specify a fragmentation scheme for the temporary table.

  The database server uses only dbspaces that are marked temporary if you specify the TEMP keyword with the DBSPACETEMP configuration parameter. If DBSPACETEMP is set with the default, NOTCRITICAL, the database server can use any dbspace except the root dbspace or a dbspace that contains logical or physical logs.

  If you specify a list of dbspaces or dbslices as an argument to the DBSPACETEMP configuration parameter, only the specified spaces are used for temporary files. If you include ordinary logging dbspaces or dbslices, these spaces can be used for temporary files created with the TEMP keyword, which are logged by default and support rollback.

  For information about explicit temporary tables, see "Explicit Temporary Tables" on page 6-8.

- When the database server writes to disk any overflow that results from the following database operations:
  - SELECT statement with GROUP BY clause
  - SELECT statement with ORDER BY clause
  - Hash join operation
  - Outer join operation
  - Index builds

Warning: *If you do not specify a value for the DBSPACETEMP configuration parameter, the database server uses the root dbspace.*

Three general guidelines apply to creating temporary dbspaces for DSS applications:

- Configure as much space as possible for temporary dbspace.

  Large DSS applications often require as much temporary disk space as permanent table and index disk space. For calculations of the minimum temporary space requirements, use the formulas for calculating table size in Chapter 6, "Table Performance."
- Specify temporary dbspaces for each coserver, and balance the temporary space across all coservers so that each coserver has the same amount of temporary space and the same number of temporary dbspaces.

  Threads running on each coserver can use only the temporary space on that coserver.
- To optimize I/O throughput, create temporary dbspaces on separate disks from the permanent dbspaces if possible.

Important: *After you create temporary dbspaces and specify them in the DBSPACETEMP configuration parameter, you must restart the database server to include the new setting.*

To create a dbspace for the exclusive use of temporary tables and sort files, use one of the following commands:

- The **onutil** CREATE TEMP DBSLICE command to create temporary dbspaces across multiple coservers
- The **onutil** CREATE TEMP DBSPACE command to create a temporary dbspace on one coserver

For best performance, use the following guidelines:

- To balance the I/O impact, place each temporary dbspace on a separate disk.
- To maximize parallel processing on each coserver node, the number of temporary dbspaces should be greater than or equal to the number of CPU VPs specified on each coserver.

The database server does not perform logical or physical logging of temporary dbspaces, and temporary dbspaces are never backed up as part of a full-system backup. You cannot mirror a temporary dbspace.

To simplify management of temporary dbspaces and improve performance, create temporary dbslices and specify the dbslice names as the setting of DBSPACETEMP. Make sure that the dbspaces in the dbslices are balanced across all coservers and all disks on each coserver. The database server distributes temporary query activity by round-robin through the temporary dbspaces. If you do not balance the dbspaces across the coservers, the coserver with more temporary dbspaces uses more of its resources for temporary file activity and might create a throughput bottleneck or a disk hot spot.

For information about how to use the **onutil** utility to create temporary dbspaces and dbslices, see the *Administrator's Reference*.

DBSPACETEMP Configuration Parameter

The DBSPACETEMP configuration parameter can specify a list of dbspaces or dbslices in which the database server places temporary tables and sort files by default. The database server fragments temporary tables across all the listed dbspaces, using a round-robin distribution scheme. (See "Designing a Distribution Scheme" on page 9-23.)

If you set the DBSPACETEMP configuration parameter to TEMP, the database server uses only the dbspaces or dbslices created with the TEMP keyword for temporary files. For detailed information about the settings of DBSPACTEMP, see the *Administrator's Reference* and "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

Important: *For best performance, use DBSPACETEMP to specify dbspaces on separate disks across all coservers for temporary tables and sort files. To simplify creating and managing temporary dbspaces, create them in dbslices that distribute the dbspaces evenly across all disks and coservers.*

DBSPACETEMP Environment Variable

The user can set the **DBSPACETEMP** environment variable to override the global DBSPACETEMP parameter. This environment variable specifies a comma- or colon-separated list of dbspaces or dbslices in which to place temporary tables for the current session.

Use the DBSPACETEMP configuration parameter with the TEMP keyword or the **DBSPACETEMP** environment variable to improve performance of sort operations and prevent the database server from unexpectedly filling file systems.

Informix recommends that you use **DBSPACETEMP** instead of the **PSORT_DBTEMP** environment variable to provide sort file space for the following reasons:

- **DBSPACETEMP** usually yields better performance.

  When dbspaces reside on character-special devices (also known as *raw* disk devices) the database server uses unbuffered disk access. I/O is faster to raw devices than to regular (buffered) operating-system files because the database server manages the I/O operation directly.

- **PSORT_DBTEMP** specifies one or more operating-system directories in which to place sort files.

  These operating-system files can unexpectedly fill on your computer because the database server does not manage them and the database server utility programs do not provide monitoring for them.

Temporary Space Estimates

Use the following guidelines to estimate the amount of temporary space to allocate:

- For OLTP applications, allocate temporary dbspaces that equal at least 10 percent of the size of all tables. If the total table size is 100 gigabytes, create at least 10 gigabytes of evenly distributed temporary dbspaces.
- DSS applications might require temporary space equal to more than 50 percent of the amount of permanent table space. If you have enough disk space, it is prudent to create at least as much temporary dbspace as permanent table space.

  DSS applications also often create explicit temporary tables to improve query processing speed. Allow for such tables in your estimates of the temporary space required.

  Although hash joins are performed entirely in shared memory if possible, hash tables might overflow to temporary space on disk. Use the following formula to estimate the amount of memory that is required for the hash table in a hash join:

  ```
  hash_table_size = (32 bytes + row_size) * num_rows_smalltab
  ```

  The value for *num_rows_smalltab* should be the number of rows in the probe table, which is the table used to probe the hash table.

I/O for Tables and Indexes

One of the most frequent functions that the database server performs is to bring data and index pages from disk into memory. The database server can read pages individually for brief transactions and sequentially for some queries. You can configure the number of pages that the database server brings into memory and the timing of I/O requests for sequential scans. You can also indicate how the database server is to respond when a query requests data from a dbspace that is temporarily unavailable.

Sequential Scans

When a query requires a sequential scan of data or index pages, most of the I/O wait time occurs while the database server seeks the appropriate starting page. If you bring in a number of contiguous pages with each I/O operation, performance for sequential scans improves dramatically. Bringing additional pages in with the first page in a sequential scan is called *read-ahead*.

The timing of the I/O operations that are required for a sequential scan is also important. If the scan thread must wait for the next set of pages to be brought in after it works its way through each batch, a delay results. Timing second and subsequent read requests to bring in pages before they are needed provides the greatest efficiency for sequential scans.

The number of pages to bring in and the frequency of read-ahead I/O requests depend on the availability of space in the memory buffers. Read-ahead can increase page cleaning to unacceptable levels if too many pages are brought in with each batch or if batches are brought in too often. For information on how to configure read-ahead, refer to "RA_PAGES, IDX_RA_PAGES, RA_THRESHOLD, and IDX_RA_THRESHOLD" on page 5-19.

Light Scans

In some circumstances, the database server bypasses the LRU queues and does not use the buffer pool when it performs a sequential scan. Such a sequential scan is termed a *light scan*.

Light scans can occur on STATIC tables at any isolation level or on other table types in the following isolation levels:

- In Dirty Read isolation level, light scans can occur for all tables, including nonlogging tables.
- In Repeatable Read isolation level, light scans can occur if the table has a shared or exclusive lock.
- In Committed Read isolation level, light scans can occur if the table has a shared or exclusive lock.
- In Cursor Stability isolation level, light scans can occur only on STATIC tables.

The performance of light scans is based on the following factors:

- Up-to-date table statistics
- The value of RA_PAGES and RA_THRESHOLD
- The size of the virtual shared-memory segment

In the **onstat -g scn** output, the **ScanType** column shows whether each thread is using a light scan or is using the buffer pool. The following **onstat - scn** output indicates that the thread is using the buffer pool to scan the table:

```
RSAM sequential scan info
SesID Thread Partnum  Rowid    Rows Scan'd Scan Type Lock Mode  Notes
16    994    30002    1cc4e    61583       Buffpool  SLock+Test
```

In the **Scan Type** column the **onstat -g scn** output might also display `Light`, for a light scan; `Keyonly`, for an index scan; and `Rids`, for an index scan that returns row identifiers. If an index scan is sending row identifiers to a light scan, `Skip scan` appears in the **Notes** column.

Similar information is displayed for the light index scans. Light index scans can occur if the index meets these requirements:

- Only one index fragment must be scanned after unnecessary index fragments are eliminated.
- The WHERE clause of the query does not contain an IN clause or OR operator.
- The query does not contain any subqueries.
- The index fragment cannot be a Generalized Key (GK) or bitmap index.

Light Appends

Light appends add table rows quickly. Loading data in Express mode from RAW tables uses light append. Light appends have the following characteristics:

- Space in existing pages is not reused. Rows are inserted in unused pages after the used pages in each table fragment. Unless you load less than one page of data, new pages are added after existing pages. If you load less than one page of data, the database server tries to add the data to an existing page.
- Rows to insert pass through large private buffers to bypass the overhead of the buffer pool.
- Insertion of these rows is not logged.

To determine if light appends occur

1. Execute **onstat -g sql** to obtain the session ID.
2. Execute **onstat -g xmp** to obtain the query ID for your session ID.
3. Execute **onstat -g xqs *queryid*** to display the SQL operators.

   Light appends are occurring if you see multiple INSERT or FLEX INSERT operators.

Unavailable Data

Another aspect of table I/O has to do with situations in which a query requests access to a table or fragment in a dbspace that is temporarily unavailable. When the database server determines that a dbspace is unavailable as the result of a disk failure, queries directed to that dbspace fail by default. You can specify dbspaces that can be skipped by queries when they are not available, as described in "DATASKIP" on page 5-20.

Many DSS queries require statistical data instead of exact data. When you specify dbspaces that can be skipped, consider the degree of precision that your queries require and how the data in the skipped dbspaces might affect query results.

Important: *If a dbspace is unavailable because a coserver is down, queries cannot be processed. All coservers must be on-line during query processing.*

Configuration Parameters That Affect I/O for Tables and Indexes

The following configuration parameters affect read-ahead:

- ISOLATION_LOCKS
- RA_PAGES and IDX_RA_PAGES
- RA_THRESHOLD and IDX_RA_THRESHOLD

In addition, the DATASKIP configuration parameter enables or disables data skipping.

The following sections describe the performance considerations associated with these parameters. For more information about database server configuration parameters, refer to the *Administrator's Reference*.

ISOLATION_LOCKS

The ISOLATION_LOCKS configuration parameter specifies the maximum number of rows that can be locked on a single scan when the Cursor Stability isolation level is in effect.

The *Cursor Stability* isolation level keeps the current row locked and does not release the lock until the row is no longer current. Use Cursor Stability when Repeatable Read is too strong because it locks the entire table, but Committed Read is too weak because it does not lock any rows. Cursor Stability allows for one or more rows to be locked but does not lock the entire table.

The default setting of ISOLATION_LOCKS is 1, but you can set it to a higher level. At some point, however, performance improvements level off, and concurrency conflicts approach those of the Repeatable Read isolation level or whole-table locking.

Important: *If you set ISOLATION_LOCKS to a value greater than 1, review your setting for the LOCKS configuration parameter. For more information on how to determine the value for the LOCKS configuration parameter, refer to* *"LOCKS" on page 4-19*.

RA_PAGES, IDX_RA_PAGES, RA_THRESHOLD, and IDX_RA_THRESHOLD

The RA_PAGES parameter specifies the number of data pages that the database server brings into memory in a single I/O operation during sequential table scans, index builds, and UPDATE STATISTICS processing. The RA_THRESHOLD parameter specifies the point at which the database server issues an I/O request to bring in the next set of data pages from disk.

The IDX_RA_PAGES and IDX_RA_THRESHOLD configuration parameters specify the read-ahead behavior for index pages. If your queries scan indexes for a large range of values, adjust the index read-ahead parameters to accommodate these scans. The default value for IDX_RA_PAGES is 4 for single-processor nodes and 8 for multiprocessor nodes. The default value for IDX_RA_THRESHOLD is `IDX_RA_PAGES / 2`.

Because most I/O wait time is used to seek the correct starting point on disk, you might increase the efficiency of sequential scans and index builds if you increase the number of contiguous pages that are brought in with each transfer. However, setting RA_PAGES AND IDX_RA_PAGES too large or RA_THRESHOLD and IDX_RA_THRESHOLD as too high a proportion of BUFFERS might result in unnecessary page cleaning to make room for pages that are not needed immediately.

In general, set RA_THRESHOLD close to the value of RA_PAGES so that the database server does not have to wait for read-ahead actions to be complete before it can use the pages. Use **onstat -P** output, as shown in the following example, to monitor read-ahead efficiency:

```
Profile
dskreads pagreads bufreads %cached dskwrits pagwrits bufwrits %cached
1468     845      2565859  99.94   80581    67220    593352   86.42
isamtot  open     start    read     write    rewrite  delete   commit
rollbk
3329106  582147   539920   478921   116870   31109    28704    2240     18
ovlock   ovuserthread ovbuff   usercpu  syscpu   numckpts flushes
0        0            0        1364.45  90.76    15       2114
bufwaits lokwaits lockreqs deadlks  dltouts  ckpwaits compress seqscans
2322     4        20002682 0        0        7        1218     12130
ixda-RA  idx-RA   da-RA    RA-pgsused lchwaits
0        0        13       13         107663
```

Add the values in the **ixda-RA**, **idx_RA**, and **da-RA** fields in the last row of the output. Compare the total to the value in the **RA-pgused** field. If the sum of the read-ahead field values is not approximately equal to the number of read-ahead pages used, reduce the setting of RA_PAGES to adjust the number of read-ahead pages.

Use the following formulas to calculate values for RA_PAGES and RA_THRESHOLD:

```
RA_PAGES = (BUFFERS * bp_fract) / (2 * large_queries) + 2
RA_THRESHOLD = (BUFFERS * bp_fract) / (2 * large_queries) - 2
```

bp_fract is the portion of data buffers to use for large scans that require read-ahead. For example, to allow large scans to take up to 75 percent of buffers, set *bp_fract* to `0.75`.

large_queries is the number of concurrent queries that require a read-ahead and that you intend to support.

To monitor read-ahead activity, use **onstat -P**. If the sum of the read-ahead columns is higher than **RA-pgused**, read-ahead is set too high. You might also notice a decrease in the read-cache rate if read-ahead is set too high. Adjust the IDX_RA_PAGES setting as appropriate to balance read-ahead for index and data pages.

DATASKIP

The DATASKIP configuration parameter allows you to specify which dbspaces, if any, queries can skip when those dbspaces are not available as the result of a disk failure. You can list specific dbspaces or turn data skipping on or off for all dbspaces.

To specify dbspaces to be skipped and turn DATASKIP off and on without restarting the database server, you can use the SQL statement SET DATASKIP. For more information, see the *Informix Guide to SQL: Syntax*.

The database server sets the sixth character in the SQLWARN array to `W` when data skipping is enabled. For more information about the SQLWARN array, refer to the *Informix Guide to SQL: Tutorial*.

Warning: *The database server cannot determine whether the results of queries are consistent when dbspaces can be skipped. If the dbspace contains a table fragment, the user who executes the query must be sure that the rows in that fragment are not needed for an accurate query result. If DATASKIP is on, queries with incomplete data might return results that are inconsistent with the actual state of the database or with similar queries run earlier or later, which might result in confusing or misleading query results.*

Background I/O Activities

Background I/O activities do not service SQL requests directly. Many of these activities are essential to maintain database consistency and other aspects of database server operation. However, they create overhead in the CPU and take up I/O bandwidth, taking time away from queries and transactions. If you do not configure background I/O activities properly, too much overhead for these activities can limit the transaction throughput of applications.

You must balance the requirements for background I/O activities in the following list:

- Checkpoints
- Logging
- Page cleaning
- Backup and restore
- Rollback and recovery

Checkpoints occur regardless of the amount of database activity although they occur more often as activity increases. Other background activities, such as logging and page cleaning, also occur more often as database use increases. Activities such as backups, restores, or fast recoveries occur only as scheduled or under exceptional circumstances.

Checkpoints, logging, and page cleaning are necessary to maintain database consistency. If checkpoints occur often and the logical logs are not large, database recovery takes less time. For this reason, a major consideration when you try to reduce the overhead for these activities is the delay that you can accept during recovery.

Another consideration is how page cleaning is performed. If pages are not cleaned often enough, an **sqlexec** thread that performs a query might not be able to find the available pages that it needs. The **sqlexec** thread then writes the buffer to disk (a *foreground write)* and waits for pages to be freed. Foreground writes impair performance and should be avoided. To reduce the frequency of foreground writes, increase the number of page cleaners or decrease the threshold for triggering a page cleaning. (See "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-28.) Use **xctl onstat -F** to monitor the frequency of foreground writes, as the following example shows:

```
Fg Writes     LRU Writes    Chunk Writes
0             103                311
address  flusher  state    data
302a5740 0        I        0        = 0X0
      states: Exit Idle Chunk Lru
```

Foreground writes should be eliminated or kept to a minimum. At checkpoints, the page cleaners indicated by the **flusher** field should be writing to chunks. Generally, for OLTP database servers should generate higher numbers in the **LRU Writes** column and DSS database servers should generate higher numbers in the **Chunk Writes** column.

For the most part, tuning background I/O activities involves striking a balance between appropriate checkpoint intervals, logging modes and log sizes, and page-cleaning thresholds. The thresholds and intervals that trigger background I/O activity often interact. Adjustments to one threshold might merely shift the performance bottleneck, not remove it.

Configuration Parameters That Affect Checkpoints

The following configuration parameters affect checkpoints:

- CKPTINTVL
- LOGFILES
- LOGSIZE
- LOGSMAX
- ONDBSPDOWN
- PHYSFILE
- USEOSTIME

CKPINTVL

The CKPTINTVL configuration parameter specifies the maximum interval between checkpoints. In most instances, fuzzy checkpoints are performed instead of full checkpoints to improve transaction throughput.

- Fuzzy checkpoints are faster than full checkpoints because the database server flushes fewer pages to disk. Because fuzzy checkpoints take less time to complete, the database server returns more quickly to processing queries and transactions. All changes to the data since the last full checkpoint or fast recovery are recorded in the logical log. In an emergency, fast recovery and rollback can return the database to a consistent state.
- Full checkpoints flush all dirty pages in the buffer pool to disk to ensure that the database is physically consistent. The database server can skip a checkpoint if all data is physically consistent at the checkpoint time.

For information about when fuzzy checkpoints are performed and when full checkpoints are performed, refer to the discussion of checkpoints in the *Administrator's Guide*.

The database server writes a message to the message log to note the time that it completes a checkpoint. To read these messages, use **onstat -m**.

Checkpoints also occur whenever the physical log becomes 75 percent full. However, with fuzzy checkpoints the physical log does not fill as rapidly because fuzzy operations in pages are not physically logged. If you set CKPTINTVL to a long interval, you can use physical-log capacity to trigger checkpoints based on actual database activity instead of at a fixed interval. Nevertheless, a long checkpoint interval can increase the time that is needed for recovery if the system fails.

Depending on your throughput and data-availability requirements, you can choose an initial checkpoint interval of 5, 10, or 15 minutes, with the understanding that checkpoints might occur more often if physical-logging activity requires them.

LOGFILES, LOGSIZE, LOGSMAX, and PHYSFILE

The LOGSIZE parameter specifies the size of the logicaL log. Use the following formula to obtain an initial estimate for LOGSIZE in kilobytes:

```
LOGSIZE = (connections * maxrows) * 512
```

connections is the number of connections for all network types specified in the **sqlhosts** file or registry by one or more NETTYPE parameters.

maxrows is the largest number of rows to be updated in a single transaction.

LOGSIZE, LOGFILES, and LOGSMAX indirectly affect checkpoints because they specify the size and number of logical-log files. Because fuzzy operations are written to the logical logs instead of being flushed to the disk, logical logs fill more rapidly with fuzzy checkpoints. If your database system creates many transactions, you probably need more logical logs than you would need if only full checkpoints were performed.

A checkpoint can occur when the database server detects that the next logical-log file to become current contains the most-recent checkpoint record. The size of the log also affects the thresholds for long transactions. The log should be large enough to accommodate the longest transaction, except as the result of an error, that you are likely to encounter. For more information, see "LTXHWM and LTXEHWM" on page 5-27.

The PHYSFILE parameter specifies the size of the physical log. This parameter indirectly affects checkpoints because whenever the physical log becomes 75 percent full, a checkpoint occurs. If your workload requires intensive updates and updates do not usually occur on the same pages, you can use the following formula to calculate a maximum size for the physical log:

```
PHYSFILE = (connections * 20 * pagesize) / 1024
```

You can reduce the size of the physical log if applications require fewer updates or if updates tend to cluster within the same data pages. If you increase the checkpoint interval, as explained in "CKPINTVL" on page 5-23, or expect increased activity, consider increasing the size of the physical log. To use physical-log fullness to trigger checkpoints, decrease the size of the physical log.

ONDBSPDOWN

The ONDBSPDOWN configuration parameter specifies the database server behavior when an I/O error indicates that a dbspace is down. By default, the database server marks any dbspace that contains no critical database server data as *down* and continues processing. Critical data includes the root dbspace, the logical log, and the physical log. You must back up all logical logs and then perform a warm restore on the down dbspace to restore access to it.

The database server halts operation whenever a disabling I/O error occurs on a nonmirrored dbspace that contains critical data, regardless of the setting for ONDBSPDOWN. In such an event, you must perform a cold restore of the database server to resume normal database operations.

When ONDBSPDOWN is set to 2, the database server continues processing to the next checkpoint and then suspends processing of all update requests. The database server repeatedly retries the I/O request that produced the error until the dbspace is repaired and the request is complete or until the database server administrator intervenes. The administrator can use **onmode -O** to mark the dbspace *down* and continue processing while the dbspace remains unavailable or use **onmode -k** to halt the database server.

Important: *If you set ONDBSPDOWN to 2, be sure to monitor the status of your dbspaces continuously.*

When ONDBSPDOWN is set to 1, the database server treats all dbspaces as though they were critical. Any nonmirrored dbspace that becomes disabled halts normal processing and requires a cold restore. The performance impact of halting the database server and performing a cold restore when any dbspace goes down can be severe.

Important: *If you decide to set ONDBSPDOWN to 1, consider mirroring all your dbspaces.*

USEOSTIME

The USEOSTIME parameter specifies whether the database server uses subsecond precision when it gets the time for SQL statements. If database applications do not require subsecond precision, set USEOSTIME to 0 to avoid unnecessary calls to the operating-system clock.

When USEOSTIME is set to 0, the database server gets the operating system clock time every second and stores the time in a shared-memory location that is accessible to all processes. When USEOSTIME is set to 1, the database server gets the operating system clock time each time a process requires the time of day.

The cost of calls to the operating system clock differs on different platforms. Nevertheless, setting USEOSTIME to 0 generally improves system performance.

Configuration Parameters That Affect Logging

The following configuration parameters affect logging:

- LOGBUFF
- PHYSBUFF
- LTXHWM
- LTXEHWM

LOGBUFF and PHYSBUFF

The LOGBUFF and PHYSBUFF configuration parameters affect logging I/O activity because they specify the respective sizes of the logical- and physical-log buffers in shared memory. The size of these buffers determines how quickly the logs fill and therefore how often they are flushed to disk.

To monitor physical and logical log buffer size and activity, use **onstat -l**. For the physical log, the output of **onstat -l** should show that the **pages/io** is approximately 75 percent of the **bufsize** column.

Logical-log buffer usage depends on whether logging is buffered. If logging is not buffered, buffer-flushing intervals depend on the size of the transactions and not on how much buffer space is available. If most transactions are smaller than the buffer page size, the ratio of **pages/io** to **bufsize** might always be low.

LTXHWM and LTXEHWM

The LTXHWM and LTXEHWM configuration parameters specify the maximum limits for long transactions and long-transaction exclusive rollbacks respectively.

The LTXHWM parameter specifies the percentage of a logical log that can fill before the database server starts to check for long transactions.

The LTXEHWM parameter specifies the point at which the database server suspends new transaction activity to locate and roll back a long transaction. These events should be rare. If they occur, they might indicate a serious problem within an application. Informix recommends a value of `50` for LTXHWM and `60` for LTXEHWM. If you decrease these thresholds, consider increasing the size of your logical-log files.

For related information, see "LOGFILES, LOGSIZE, LOGSMAX, and PHYSFILE" on page 5-24.

Configuration Parameters That Affect Page Cleaning

The following configuration parameters affect page cleaning:

- CLEANERS
- LRUS
- LRU_MAX_DIRTY
- LRU_MIN_DIRTY
- RA_PAGES
- RA_THRESHOLD

"RA_PAGES, IDX_RA_PAGES, RA_THRESHOLD, and IDX_RA_THRESHOLD" on page 5-19 describes the RA_PAGES and RA_THRESHOLD parameters.

CLEANERS

The CLEANERS configuration parameter specifies the number of page-cleaner threads to run.

For installations that support fewer than 20 disks, Informix recommends one page-cleaner thread for each disk that contains database server data. For installations that support between 20 and 100 disks, Informix recommends one page-cleaner thread for every two disks. For larger installations, Informix recommends one page-cleaner thread for every four disks. If you increase the number of LRU queues as the next section describes, increase the number of page-cleaner threads proportionally.

LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY

The LRUS configuration parameter specifies the number of least recently used (LRU) queues to set up within the shared-memory buffer pool. The buffer pool is distributed among LRU queues. Configuring more LRU queues allows more page cleaners to operate and reduces the size of each LRU queue. For a single-processor system, Informix suggests that you set the LRUS parameter to a minimum of 4. For multiprocessor systems, set the LRUS parameter to a minimum of 4 or NUMCPUVPS, whichever is greater.

The setting of LRUS is combined with LRU_MAX_DIRTY and LRU_MIN_DIRTY to control how often pages are flushed to disk between full checkpoints. In some cases, you can achieve high throughput by setting these parameters so that very few modified pages remain to be flushed when checkpoints occur. The main function of the checkpoint is then to update the physical-log and logical-log files.

To monitor the percentage of dirty pages in LRU queues, use **xctl onstat -R**. When the number of dirty pages consistently exceeds the LRU_MAX_DIRTY limit, you have too few LRU queues or too few page cleaners. First use the LRUS parameter to increase the number of LRU queues. If the percentage of dirty pages still exceeds LRU_MAX_DIRTY, use the CLEANERS parameter to increase the number of page cleaners.

Configuration Parameters That Affect Fast Recovery

The following configuration parameters affect fast recovery:

- OFF_RECVRY_THREADS
- ON_RECVRY_THREADS

The OFF_RECVRY_THREADS and ON_RECVRY_THREADS parameters specify the number of recovery threads that operate when the database server performs a cold restore, warm restore, or fast recovery. The OFF_RECVRY_THREADS configuration parameter specifies the number of threads for cold restore. The ON_RECVRY_THREADS configuration parameter specifies the number of threads for warm restore and fast recovery.

The number of threads should usually match the number of tables or fragments that are frequently updated on each coserver to roll forward the transactions recorded in the logical log.

Another way to estimate the threads required is to use the number of tables or fragments that are frequently updated. On a single-CPU coserver, the number of threads should be no fewer than 10 and no more than 30 or 40. At a certain point, however, the overhead that is associated with each thread outweighs the advantages of parallel threads.

A warm restore takes place concurrently with other database operations. To reduce the impact of the warm restore on other users, you can allocate fewer threads to it than you would to a cold restore.

Chapter

6

Table Performance

In This Chapter

This chapter describes performance considerations that are associated with unfragmented tables and with table fragments and indexes.

This chapter discusses the following issues:

- Table types and their performance implications
- Table placement on disk to increase throughput and reduce contention
- Estimating table size
- Changing table definitions
- Denormalizing the database for improved performance

An instance of Extended Parallel Server consists of one or more *shared-nothing* coservers. This shared-nothing architecture means that performance tuning of dbspaces, temporary space, and tables should be approached both from the local coserver level and from the global database server level.

The topics discussed in this chapter focus on general table questions, such as choosing table types and creating indexes on appropriate columns, as well as other local coserver tuning issues, such as monitoring interleaved extents:

- For information about table fragmentation for enhanced parallel processing, refer to Chapter 9, "Fragmentation Guidelines."
- For table I/O performance guidelines, refer to "I/O for Tables and Indexes" on page 5-14.

As explained in "Maintenance of Good Performance" on page 1-35, maintaining good database performance is the joint responsibility of the operating-system administrator, the database administrator, and the database and application designers. However, the database administrator or the database and application designers can handle most of the tuning issues that this chapter discusses. Only tuning measures that involve changes to physical structures, such as disk volumes and chunks, require the assistance of the operating-system administrator.

Choosing Table Types

Extended Parallel Server provides several table types for various purposes. The table type that you create depends on the kind of queries and transactions that are run on the tables. Each table type is designed for a specific use.

The STANDARD type, which corresponds to a table in a logged database, is the default. If you issue the CREATE TABLE statement without specifying the table type, you create a STANDARD table.

Figure 6-1 lists the available types of tables and their general features.

Figure 6-1
Available Table Types

Type	Permanent	Logged	Indexes	Light Append	Rollback
RAW	Yes	No	No	Yes	No
STATIC	Yes	No	Yes	No	No
OPERATIONAL	Yes	Yes	Yes	Yes †	Yes
STANDARD	Yes	Yes	Yes	No	Yes
TEMP	No	Yes*	Yes	Yes	Yes*
SCRATCH	No	No	No	Yes	No

† If no triggers are defined on the table

* If created in a logging dbspace, not a temporary dbspace

You can use the ALTER TABLE statement to change a permanent table type, with the following restrictions:

- Before you convert a table type to RAW, you must drop all indexes and constraints.
- Before you convert a table to STANDARD, you must perform a level-0 backup.

You cannot alter a TEMP or SCRATCH table or convert a permanent table to a temporary table.

Using STANDARD Tables

STANDARD tables are the default table type. They permit full logging and recovery from backups, as well as all insert, delete, and update operations.

STANDARD tables do not permit light append, which is used for efficient bulk loading of data. Data in STANDARD tables must be loaded row by row instead. This feature is not a disadvantage in a real-time OLTP environment in which users update or add rows one or a few at a time. Because STANDARD tables are logged, the recoverability provided by the rollback and fast recovery capabilities means that data changes for these tables cannot be lost. Before you convert any modified nonlogging table to STANDARD type, make a level-0 backup of the table.

You need to set the lock mode appropriately for STANDARD tables, depending on how they are used, and also to make sure that SELECT statements and update cursors are executed at an appropriate isolation level. For information about setting lock modes for tables, see "Specifying a Table Lock Mode" on page 6-9. For descriptions of the isolation levels and their use, see "Setting the Isolation Level" on page 8-9.

Using RAW Tables

RAW tables are used primarily to load data from external tables. RAW tables use light appends, but they are not logged and do not support indexes, referential constraints, or rollback. Whether fast recovery or restoration from a backup is possible depends on several factors. For more information, refer to the chapters in the *Administrator's Guide* that refer to data storage locations and fast recovery.

For extremely fast data loading from an external table in express mode, use RAW tables to use light append and eliminate constraint-checking and index-building overhead. Light-append operations append rows to a table and bypass the buffer cache to eliminate buffer-management overhead. Because rows are appended to the table, existing space in table pages is not reused. The table is locked exclusively during an express load so that no other user can access the table during the load. If you update tables by deleting old data and loading new data, use deluxe mode to reuse the space in the table.

After a RAW table is created from loaded data, use the ALTER TABLE statement to convert the table from RAW to a different type. Before you convert a RAW table to a STANDARD table, perform a level-0 backup.

Using STATIC Tables

STATIC tables permit read-only access and for this reason are not logged. Use them for stable data, such as postal codes, city and state combinations, department numbers and names, and so on. Because STATIC tables are read-only, they are not locked when they are accessed, which reduces overhead and speeds queries and transactions.

Although inserts, updates, or deletes are not permitted, indexes and referential constraints are allowed. To avoid locking overhead and contention problems, the database server can use light scans for STATIC tables, as described in "Light Scans" on page 5-15.

Only STATIC tables can have Generalized Key (GK) indexes. GK indexes can store derived data or can be selective indexes that contain keys for only a subset of a table, virtual column indexes that contain the result of an expression, and join indexes that contain keys that result from joining one or more tables. The database server can use these indexes to satisfy queries instead of accessing the tables. If the database server uses a GK index on a table to process a query, however, it cannot use any other indexes on that table.

If you change the type of a table from STATIC to any other type, you must drop all GK indexes on it.

When a transaction accesses a STATIC table, Dirty Read isolation is appropriate because the table is read-only and its contents does not change.

Using OPERATIONAL Tables

OPERATIONAL tables are often used for data derived from another source so that restorabilty is not important. OPERATIONAL tables are frequently used to create tables for DSS applications by loading data from active OLTP database systems.

If an OPERATIONAL table does not have any indexes, the database server can perform express-mode loads that use light append. Light appends, however, reduce the recoverability and restorability of the table. For information about light appends, see "Light Appends" on page 5-17.

Although OPERATIONAL tables permit updates, inserts, and deletes, and are logged to permit rollback or recovery after a disk or other system failure, they can be recovered from a backup only if data was not loaded with light append after the most recent level-0 backup.

Deluxe-mode loads, which can update existing indexes, can be performed on OPERATIONAL tables if the cost of completely rebuilding table indexes is too high. Unique constraint checking is allowed. For detailed information about high-performance loading with external tables, refer to the *Administrator's Reference*.

Using Temporary Tables

Use temporary tables to reduce sorting scope, to select an ordered subset of table rows that are required by more than one query, or to create other tables that can easily be derived from permanent table data. Although temporary tables are session specific, queries run by the same session can use temporary tables that the session created earlier. When the session exits, all temporary tables are dropped.

The two types of temporary tables are implicit and explicit.

Implicit Temporary Tables

When the database server processes queries that contain clauses such as GROUP BY or when it creates an index, it automatically creates an *implicit* temporary table.

You do not control the location of implicit temporary tables. The database server places implicit temporary tables in the dbspaces or dbslices that DBSPACETEMP specifies. If your application requires large temporary files, make sure that you create temporary dbspaces or dbslices and specify these spaces as arguments to the DBSPACTEMP configuration parameter or environment variable. The default behavior of DBSPACETEMP is to use any noncritical dbspaces.

When you create dbslices or dbspaces for temporary use, make sure that space is evenly distributed across coservers and use the guidelines in "Temporary Space Estimates" on page 5-14 to calculate a total amount of space that is adequate for your applications. For information about setting DBSPACETEMP, see "DBSPACETEMP Configuration Parameter" on page 5-12.

Explicit Temporary Tables

You create an *explicit* temporary table with the CREATE TABLE statement or the SELECT...INTO statement.

Explicit temporary tables can be either logged or unlogged. Unlogged tables avoid the overhead of logging, but they cannot be recovered by roll back.

- To create an explicit temporary table that the database server can fragment efficiently for parallel processing across temporary dbspaces, use the SCRATCH option of the CREATE TABLE statement or the INTO SCRATCH clause of the SELECT statement.

  ```
  SELECT * FROM customer INTO SCRATCH temp_table
     WHERE custno > 3500
  ```

  Temporary tables created with the SELECT...INTO SCRATCH statement are not logged and cannot be indexed. If you do not need to roll back data in a temporary table and do not need indexes on the table, use SELECT...INTO SCRATCH statements or nonlogging temporary tables for best performance.

  If you create a temporary table with the CREATE SCRATCH TABLE statement, you can specify a fragmentation scheme.

- To create an explicit temporary table that can be indexed and can have defined constraints, use the CREATE TEMP TABLE statement or the INTO TEMP clause of the SELECT statement.

  TEMP tables can be fragmented explicitly. They are also logged unless you use the WITH NO LOG clause. The following SQL statements create a nonlogging temporary table that is fragmented by hash in a temporary dbslice:

  ```
  CREATE TEMP big_bills (cust_no CHAR(6), fname
  CHAR(15), lname CHAR(25), addr1 CHAR(30)...)
      WITH NO LOG
      FRAGMENT BY HASH(cust_no)
      IN dbsl1_temp;
  ```

  To populate the temporary table, use the following SELECT statement:

  ```
  SELECT * FROM customer INTO big_bills
      WHERE bill_amt > 1500;
  ```

  To create a logging TEMP table you must either specify a nontemporary dbspace or dbslice where it should be created or include nontemporary logging dbspaces or dbslices in the argument to the DBSPACETEMP configuration parameter. You cannot create a logging TEMP table in a temporary dbspace.

Explicit temporary tables and indexes on temporary tables are session specific and are visible only to the session that creates them. When the session ends, its temporary tables and indexes are removed.

For detailed information about the syntax used to create temporary tables, refer to the *Informix Guide to SQL: Syntax*.

To learn about performance advantages of explicit temporary tables, refer to "Fragmenting Temporary Tables" on page 9-58 and "Using Temporary Tables to Reduce Sorting Scope" on page 13-34. For information about the syntax of the CREATE TABLE and SELECT statements, refer to the *Informix Guide to SQL: Reference*.

Specifying a Table Lock Mode

The lock mode of a table determines what portion of a table is locked when any row is accessed. The default lock mode is PAGE, which locks the data page that contains the accessed row until the transaction is complete.

When you create or alter a table, you can set its lock mode to TABLE, PAGE, or ROW:

- If you use a table primarily for OLTP applications, you might create the table with lock mode set to ROW, so that as many clients as possible can access the table, even if they access rows on the same data page.
- If you use a table primarily for DSS queries, you might set the lock mode to TABLE to reduce the overhead of acquiring locks at a smaller granularity. However, if you run more than one DSS query against the same data, deadlocks might result.

For more information about the table lock mode and its implications for OLTP and DSS applications, see Chapter 8, "Locking."

Tip: *To lock more than one row when Cursor Stability is in effect, set ISO_CURLOCKS to a number greater than 1. If you set ISO_CURLOCKS to a number greater than 1, row buffering is more efficient, but concurrency might be reduced.*

Monitoring Table Use

The first time that the database server accesses a table, it retrieves necessary system-catalog information for the table from the disk. For each table access, this system-catalog information is stored in memory in the data-dictionary cache.

The database server still places pages for system catalog tables in the buffer pool as it does all other data and index pages. However, the data-dictionary cache offers an additional performance advantage because the data-dictionary information is organized in a more efficient format and organized to allow fast retrieval.

The database server uses a hashing algorithm to store and locate information in the data-dictionary cache. DD_HASHSIZE and DD_HASHMAX control the size of the data-dictionary cache. To specify the number of buckets in the data-dictionary cache, change the DD_HASHSIZE configuration parameter. To specify the number of tables that can be stored in one bucket, change the DD_HASHMAX configuration parameter.

For example, if DD_HASHMAX is 10 and DD_HASHSIZE is 100, you can potentially store information for 1000 tables in the data-dictionary cache, and each hash bucket can have a maximum of 10 tables. If the bucket reaches the maximum size, the database server uses a least-recently used mechanism to clear entries from the data dictionary.

To monitor data-dictionary cache activity, use the **onstat -g dic** command.

Figure 6-2
onstat -g dic Output Sample

```
Dictionary Cache (251 x 50):
Flags:
  1 := T=temp R=raw C=static O=operational E=external S=standard V=view
  2 := D=dirty
  3 := X=locked
  4 := F=Fragmented L=Fragment-Locally

List#  Siz  Flags  Refcount    Lid Memory-Stats  Table Name
            1234  Loc.Use Rem       Blk size
------------------------------------------------------------
  3    1    S--F         0.0   1          515    7 7264      sysmaster:informix.sysptnhdr
  8    1    S---          0.0   0           265    2 2048      stores_demo:informix.syssynonyms
```

The **onstat -g dic** output includes the following information.

Column Name	Description
List#	Bucket number
Size	Number of tables in the bucket
Flags 1234	Information about the status of the table, as indicated by the list of flag codes

(1 of 2)

Column Name	Description
Refcount: Loc and Rem	Number of current references to a specific data-dictionary cache entry A single table can have more than one data-dictionary cache entry, and more than one entry can be accessed over time and even at the same time. The **Loc** column displays the number of times that the data-dictionary cache entry has been referenced from the local coserver. The **Rem** column is applicable only for coserver 1 and indicates how many other coservers have a copy of this cache entry.
Table name	Name of the table that the data-dictionary information describes

(2 of 2)

Specifying Table Placement

Tables that the database server supports reside on one or more portions of a disk or disks in a dbspace. When you configure chunks and allocate them to dbspaces, make sure that you provide enough chunks for all of the tables or table fragments that the dbspaces will contain. To estimate the size of a table, follow the instructions in "Estimating Table Size" on page 6-15.

The way that tables are fragmented across dbspaces is a more important performance factor than the placement of the table on the physical disk. For information about fragmentation issues, see Chapter 9, "Fragmentation Guidelines."

Assigning Tables to Dbspaces

When you create a table, you assign it to a dbspace in one of the following ways:

- Explicitly, with the IN DBSPACE or IN DBSLICE clause of the CREATE TABLE statement
- By default, in the dbspace of the current database

  The current database is set by the most-recent DATABASE or CONNECT statement that the DBA issues before issuing the CREATE TABLE statement.

You can fragment a table across multiple dbspaces either as individual dbspaces or grouped in dbslices, as described in "Planning a Fragmentation Strategy" on page 9-6.

Moving Tables and Table Fragments to Other Dbspaces

Use the ALTER FRAGMENT statement to move a table or table fragment to another dbspace or dbslice. The ALTER FRAGMENT statement is the simplest way to change the location of a table. However, the table is not available while the database server moves it. Schedule the movement of a table or fragment at a time that affects the fewest users. For a description of the ALTER FRAGMENT statement, refer to the *Informix Guide to SQL: Syntax*.

The database server administrator can perform the same actions with a series of SQL statements and external tables for parallel inserts, as described in "Loading and Unloading Tables" on page 6-31 and the *Administrator's Reference*. Other methods for moving tables to different dbspaces might be simpler. To unload the data from a table and then move that data to another dbspace, use the LOAD and UNLOAD statements as described in the *Informix Guide to SQL: Syntax*.

When you move a table between databases with LOAD and UNLOAD or other SQL statements, the table can become inconsistent with the rest of the database while data from the table is copied and reloaded. To prevent inconsistency in databases that are updated by user input, restrict access to the version that remains on disk while the data transfer occurs. If you use the loaded tables for a read-only DSS-query database, minor inconsistencies might not be important.

Depending on the size and fragmentation strategy of the table, and its associated indexes, it might be faster to unload a table and reload it than to alter fragmentation. For other tables, it might be faster to alter fragmentation. You might have to experiment to determine which method is faster for moving or repartitioning a table.

Managing High-Use Tables

High-use tables require special management for performance, especially in OLTP applications. The database server provides two levels of management for such tables.

If a table or table fragment has high I/O activity, place it in a dbspace on a dedicated disk to reduce contention for the data that is stored in that table. When disk drives have different performance levels, you can put the tables with the highest use on the fastest drives. Placing two high-use tables on separate disks reduces competition for disk access when the two tables experience frequent, simultaneous I/O from multiple applications or when joins are formed between them.

To isolate a high-use table on a separate disk, assign the entire disk to a chunk. The system administrator creates chunks on each disk in such a way that the chunks can be associated with a specific coserver. If chunks are not assigned to an entire disk but are located at offsets, make sure that only one coserver accesses all of the chunks on the disk.

For information about fragmenting tables across coservers, see "Designing a Distribution Scheme" on page 9-23.

Improving Table Performance

The following factors affect the performance that is associated with a table or table fragment:

- The placement of the table on the disk, as the previous sections describe
- The size of the table or fragment
- The indexing strategy that you use
- The size of table extents and how they are grouped
- The frequency of table access

Estimating Table Size

This section describes how to calculate the approximate number of disk pages required for tables.

After you have estimated the total table size, refer to "Planning a Fragmentation Strategy" on page 9-6 for information about fragmenting the table across the coservers in your database server.

The disk pages that are allocated to a table or table fragment are collectively referred to as a *tblspace*. Attached index pages and pages that store BYTE or TEXT data are stored in separate tblspaces in the dbspace that contains the associated data pages of the table.

The tblspace does not correspond to any single physical region in the chunks assigned to the dbspace. The data extents and indexes that make up a table can be scattered through the dbspace.

The following sections describe how to estimate the page count for each type of page associated with a table.

Tip: *If an appropriate sample table already exists, or if you can use simulated data to build a sample table of realistic size, you do not need to estimate. You can run* ***onutil*** *CHECK INFO IN TABLE to obtain exact numbers.*

Estimating Data Page Size

How you estimate the data pages of a table depends on whether the length of the table rows is fixed or variable.

Estimating the Size of Tables with Fixed-Length Rows

Perform the following steps to estimate the number of disk pages required for a table with fixed-length rows. A table with fixed-length rows has no columns of type VARCHAR or NVARCHAR.

To estimate the page size, row size, number of rows, and number of data pages

1. Subtract 60 bytes from the page size for the header on each data page. The resulting amount is referred to as *pageuse*.

   The page size can be set to 2048, 4096, or 8092 bytes, as described in "PAGESIZE" on page 4-21.

2. To calculate the size of a row, add the widths of all the columns in the table definition. TEXT and BYTE columns each use 56 bytes. If you have already created the table, use the following SQL statement to obtain the size of a row:

   ```
   SELECT rowsize FROM systables
      WHERE tabname = 'table-name';
   ```

3. Estimate the number of rows that the table should contain. This number is referred to as *rows*.

   The procedure for how to calculate the number of data pages that a table requires differs depending on whether the row size is less than or greater than *pageuse*.

4. If the size of the row is less than or equal to *pageuse*, use the following formula to calculate the number of data pages. The **trunc()** function notation indicates that you should round down to the nearest integer.

   ```
   data_pages = rows / trunc(pageuse/(rowsize + 4))
   ```

   The maximum number of rows per page is 255, regardless of the size of the row.

5. If the size of the row is greater than *pageuse*, the database server divides the row among pages. The page that contains the initial portion of a row is called the *home page*. Pages that contain subsequent portions of a row are called *remainder pages*. If a row spans more than two pages, some of the remainder pages are completely filled with data from that row. When the trailing portion of a row uses less than a page, it can be combined with the trailing portions of other rows to fill out the partial remainder page. The number of data pages is the sum of the home pages, the full remainder pages, and the partial remainder pages.

   a. Calculate the number of home pages. The number of home pages is the same as the number of rows.

      ```
      homepages = rows
      ```

   b. Calculate the number of full remainder pages. First, to calculate the size of the row remainder, use the following formula:

      ```
      remsize = rowsize - (pageuse + 8)
      ```

      If *remsize* is less than *pageuse* - 4, you have no full remainder pages. If *remsize* is greater than *pageuse* - 4, you can use *remsize* in the following formula to obtain the number of full remainder pages:

      ```
      fullrempages = rows * trunc(remsize/(pageuse - 8))
      ```

   c. Calculate the number of partial remainder pages. First calculate the size of a partial row remainder left after the home and full remainder pages for an individual row have been accounted for. In the following formula, the **remainder()** function notation indicates that you should take the remainder after division:

      ```
      partremsize = remainder(rowsize/(pageuse - 8)) + 4
      ```

      The database server uses page-size thresholds to determine how many partial remainder pages to use. If *remsize* is greater than *pageuse* - 4, use the following formula to calculate the ratio of the partial remainder to the page:

      ```
      partratio = partremsize/pageuse
      ```

If *remsize* is less than *pageuse* - 4, use *remsize* instead of *partremsize*. in the formula.

Use the appropriate formula from the following chart to calculate the number of partial remainder pages (*partrempages*) by using the value of *partratio* that you obtain.

Partratio	Formula to calculate the number of partial remainder pages
Less than .1	partrempages = *rows* / (trunc((*pageuse* / 10) / *remsize*) + 1)
Less than .33	partrempages = *rows* / (trunc((*pageuse* / 3) / *remsize*) + 1)
.33 or larger	partrempages = *rows*

d. To calculate the total number of pages, use the following formula:

```
tablesize = homepages + fullrempages + partrempages
```

Important: *Although the maximum size of a row that the database server accepts is approximately 32 kilobytes, performance deteriorates when a row exceeds the size of a page. For information on breaking up wide tables for improved performance, refer to* *"Denormalizing the Data Model to Improve Performance" on page 6-40*.

Estimating the Size of Tables with Variable-Length Rows

When a table contains one or more VARCHAR or NVARCHAR columns, its rows usually have varying lengths that reduce the precision of the calculations. You must estimate the typical size of each VARCHAR column, based on your understanding of the data, and use that value when you make your estimates.

Important: *When the database server allocates space to rows of varying size, it considers a page to be full when it does not contain enough space for an additional row of the maximum size.*

To estimate the size of a table with variable-length rows, make the following estimates and choose a value between them, based on your understanding of the data:

- The maximum size of the table, calculated with the maximum width allowed for all VARCHAR or NVARCHAR columns
- The projected size of the table, calculated with a typical width for each VARCHAR or NVARCHAR column

The database server stores the size of the column in an extra byte that is added to each variable-length column.

To estimate the maximum number of data pages

1. To calculate *rowsize*, add together the maximum values for all column widths.
2. Use this value for *rowsize* and perform the calculations described in "Estimating the Size of Tables with Fixed-Length Rows" on page 6-16. The resulting value is called *maxsize*.

To estimate the projected number of data pages

1. To calculate *rowsize*, add together typical values for each of your variable-width columns. Informix suggests that you use the most frequently occurring width in a column as the typical width for that column. If you do not have access to the data or do not want to tabulate widths, you might choose a fraction of the maximum width, such as two-thirds.
2. Use this value for *rowsize* and perform the calculations described in "Estimating the Size of Tables with Fixed-Length Rows" on page 6-16. The resulting value is called *projsize*.

Selecting an Intermediate Value for the Size of the Table

The actual table size should fall somewhere between *projsize* and *maxsize*. Based on your knowledge of the data, choose a value in that range that seems reasonable to you. The less familiar you are with the data, the higher your estimate should be.

Estimating Dbspace Pages for Simple Large Objects

When you estimate space required for a table, include space for simple large objects that are to be stored in that dbspace. Simple large objects include only TEXT and BYTE data.

To estimate the number of pages for simple large objects

1. Calculate the usable portion of the page with the following formula:

   ```
   bpuse = 4096 - 32
   ```

   The 4096 bytes is the default size of a page. If you have set the PAGESIZE configuration parameter to a different number, as described in "PAGESIZE" on page 4-21, use that number. The 32 bytes is for overhead.

2. For each simple large object of size *n*, calculate the number of pages that the simple large object occupies (*bpages_n*) with the following formula:

   ```
   bpages1 = ceiling(bsize1/bpuse)
   bpages2 = ceiling(bsize2/bpuse)
   ...
   bpages_n = ceiling(bsize_n/bpuse)
   ```

   The **ceiling()** function notation indicates that you should round up to the nearest integer value.

3. Add up the total number of pages for all simple large objects, as follows:

   ```
   total_bpages = bpages1 + bpages2 + ... + bpages_n
   ```

Alternatively, you can base your estimate on the median size of a simple large object, which is the large object size that occurs most frequently. This method is less precise, but it is easier to calculate.

To estimate the number of pages based on the median size of the simple large objects

1. Average the sizes of the TEXT or BYTE data to obtain the median size of a simple large object.

   ```
   msize = avg(bsize1 + bsize2 +
               ... + bsizen) / n
   ```

2. Calculate the number of pages required for a simple large object of median size as follows:

   ```
   mpages = ceiling(msize/bpuse)
   ```

3. Multiply this amount by the total number of simple large objects, as follows:

   ```
   total_bpages = bcount * mpages
   ```

Managing Extents

As you add rows to a table, the database server allocates disk space to it in units called *extents*. Each extent is a block of physically contiguous pages from the dbspace. Even if the dbspace includes more than one chunk, each extent is allocated entirely in a single chunk so that its pages can remain contiguous.

Contiguity is important to performance. When the data pages are contiguous, disk-arm motion is minimized when the database server reads the rows sequentially. The use of extents is a compromise between the following conflicting factors:

- Most dbspaces are shared among several tables.
- The size of some tables is not known in advance.
- Tables grow at different times and different rates.
- All the pages of a table should be adjacent for best performance.

Because table sizes are not always predictable, table space cannot be preallocated. The database server adds extents only when they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when the database server creates a new extent adjacent to the previous extent, it treats both extents as a single extent.

Choosing Extent Sizes

When you create a table, you can specify the size of the first extent as well as the size of each extent to be added as the table grows. The following example creates a table with a 512-kilobyte initial extent and 100-kilobyte next extents:

```
CREATE TABLE big_one (…column specifications…)
    IN big_space
    EXTENT SIZE 512
    NEXT SIZE 100
```

The default value for the extent size and the next-extent size is eight times the disk page size on your system. If your system uses the default page size of 4-kilobytes, the default extent size and the next-extent size is 32 kilobytes.

To change the next-extent size, use the ALTER TABLE statement. This change has no effect on extents that already exist. The following example changes the next-extent size of the table to 50 kilobytes:

```
ALTER TABLE big_one MODIFY NEXT SIZE 50
```

When you fragment an existing unfragmented table, you should also reduce its next-extent size because each fragment requires less space than the original table. If the unfragmented table was defined with a large next-extent size, the database server uses that same size for the next-extent on *each* fragment, which results in over-allocation of disk space.

For example, if you fragment the preceding **big_one** sample table across five coservers, you can alter the next-extent size to one-fifth the original size. For more information on the ALTER FRAGMENT statement, see the *Informix Guide to SQL: Syntax*. The following example changes the next-extent size to one-fifth of the original 100-kilobyte size:

```
ALTER TABLE big_one MODIFY NEXT SIZE 20
```

The next-extent sizes of the following kinds of tables are not as important for performance:

- A small table is a table that requires only one extent. If such a table is heavily used, large parts of it remains in a memory buffer.
- An infrequently used table is not important to performance no matter what size it is.
- A table in a dedicated dbspace is always allocated new extents that are adjacent to its old extents. The size of these extents is not important because, being adjacent, they are treated as one large extent.

  If the table has an index and the index is stored in the same dbspace, however, newly allocated extents probably are not contiguous.

When you assign an extent size to these kinds of tables, the only consideration is to avoid creating large numbers of extents. The more extents that a table occupies, the longer the database server takes to find the data. In addition, an upper limit exists on the number of extents allowed, as explained in "Limiting the Number of Extents for a Table" on page 6-24.

The only limit to the size of an extent is the size of the chunk. If you expect tables in a dbspace to grow steadily to an unknown size, assign next-extent sizes large enough so that the dbspace will use a small number of extents.

The following steps suggest one approach to assigning extents for tables of unknown size.

To allocate space for table extents

1. Decide how to allocate space among the tables in the dbspace. For example, you might divide the dbspace among three tables so that one table has 40 percent, another table has 20 percent, and a third table has 30 percent, with 10 percent reserved for small tables and overhead.
2. Assign each table a quarter of its share of the dbspace as its initial extent.
3. Assign each table an eighth of its share as its next-extent size.
4. Monitor the growth of the tables regularly with **onutil**. For more information on how to use the **onutil** utility, refer to the *Administrator's Reference*.

If you do not have enough contiguous space to create an extent of the specified size as the dbspace fills up, the database server allocates as large an extent as it can.

Limiting the Number of Extents for a Table

The number of extents that a table can acquire is limited, depending on the page size and the table definition.

To calculate the upper limit on extents for a particular table, use the following set of formulas:

```
vcspace   = 8  * vcolumns + 136
tcspace   = 4  * tcolumns
ixspace     = 12 * indexes
ixparts   = 4  * icolumns
extspace  = pagesize – (vcspace + tcspace + ixspace + ixparts +
84)
maxextents = extspace/8
```

vcolumns is the number of columns that contain simple-large-object and VARCHAR data.

tcolumns is the number of columns in the table.

indexes is the number of indexes on the table.

icolumns is the number of columns named in those indexes.

pagesize is the size of a page reported by **onutil** CHECK RESERVED.

The table can have no more than *maxextents* extents.

The database server performs the following actions to help ensure that it does not exceed the table limits:

- The database server keeps track of the number of extents assigned to a table. On every sixteenth next-extent assignment, it doubles the next-extent size for the table. For example, the thirty-second next-extent is four times the size of the first next-extent.
- When the database server creates a new extent adjacent to the previous extent, it treats both extents as a single extent.

If you receive error -136 (`No more extents`) after an INSERT request, the database server needs to add an extent to a table but cannot do so. Either not enough disk space is available in the dbspace, or the table has been given the maximum number of extents that is allowed. For information about how to correct the problem, refer to *Informix Error Messages* in Answers OnLine.

Tip: *If tables overflow the allocated dbspaces, use the* ***onutil*** *ALTER DBSLICE...ADD DBSPACE command to add dbspaces to a dbslice. Then use the SQL statement ALTER FRAGMENT to redistribute the table and index fragments across all dbspaces in the dbslice.*

Checking for Extent Interleaving

When two or more growing tables share a dbspace, extents of one tblspace might be placed between extents that another tblspace uses. Such extents are *interleaved*. Interleaving creates distances between the extents of a table, as Figure 6-3 shows. Performance suffers when disk seeks for a table must skip over extents, particularly for sequential scans. Try to optimize the table-extent sizes, which limits head movement, or consider placing tables in separate dbspaces.

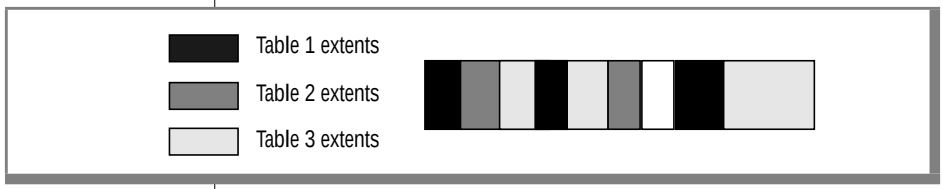


Figure 6-3
Interleaved Table Extents

To check for extent interleaving, monitor chunks. Execute **onutil** with the CHECK SPACE option to obtain the physical layout of information in the chunk. The following information appears:

- Dbspace name and owner
- Number of chunks in the dbspace
- Sequential layout of tables and free space in each chunk
- Number of pages dedicated to each table extent or free space

This output is useful for determining the degree of extent interleaving. If the database server cannot allocate an extent in a chunk despite an adequate number of free pages, the chunk might be badly interleaved.

Eliminating Interleaved Extents

You can eliminate interleaved extents with one of the following methods:

- Reorganize the table by unloading it into an external table and reloading it from the external table.
- Reorganize the tables with the UNLOAD and LOAD statements.
- Use the CREATE CLUSTER INDEX statement.
- Use the ALTER TABLE statement.

For a discussion of ways to prevent extent interleaving, refer to "Choosing Extent Sizes" on page 6-22.

Reorganizing Dbspaces and Tables to Eliminate Extent Interleaving

You can eliminate interleaved extents by rebuilding a dbspace, as Figure 6-4 illustrates. The order of the reorganized tables in the dbspace is not important. All that matters is that the extents of each reorganized table are contiguous.

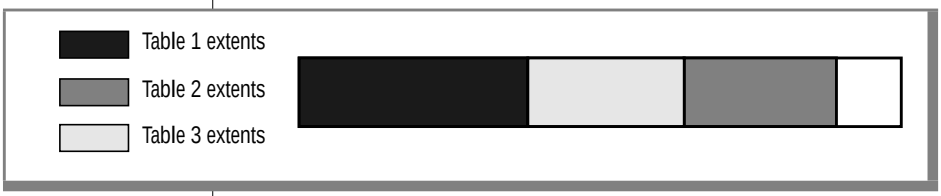


Figure 6-4
A Dbspace That Eliminates Interleaved Extents

To reorganize tables in a dbspace

1. Drop indexes to unload or reload tables rapidly.
2. Copy the tables in the dbspace to external tables individually with any of the following methods:
   - The SELECT INTO EXTERNAL TABLE statement
   - The CREATE EXTERNAL TABLE and the INSERT INTO EXTERNAL TABLE statements
   - The UNLOAD statement

   Use external tables for the best performance.

3. Drop all the tables in the dbspace.
4. Re-create and load the tables with one of the following methods, depending on how you unloaded the data:
   - The CREATE TABLE and INSERT FROM EXTERNAL TABLE statements
   - The LOAD statement

     The LOAD statement re-creates the tables with the same properties as they had before, including the same extent sizes.
5. Perform a level-0 backup, re-create indexes and run UPDATE STATISTICS on the re-created tables.

For further information about using external tables to load and unload large tables, refer to the *Administrator's Reference*. For more information about the syntax of the UNLOAD and LOAD statements, refer to the *Informix Guide to SQL: Syntax*.

Creating a Cluster Index to Eliminate Extent Interleaving

Sometimes you can eliminate extent interleaving if you create a clustered index. When you use the CLUSTER keyword of the CREATE INDEX statement, the database server sorts and reconstructs the table. The CLUSTER keyword causes the database server to reorder rows in the physical table to match the order in the index.

In Extended Parallel Server, however, you cannot create a clustered index on a STANDARD type table. For more information, refer to "Clustering Indexes" on page 7-18.

The CLUSTER keyword eliminates interleaved extents in the following circumstances:

- The chunk must contain enough contiguous space to rebuild the table.
- The database server must use this available contiguous space to rebuild the table.

  If blocks of free space exist before this larger contiguous space, the database server might allocate the smaller blocks first. The database server allocates space for the CREATE INDEX process from the beginning of the chunk, looking for blocks of free space that are greater than or equal to the size that is specified for the next extent. When the database server rebuilds the table with the smaller blocks of free space that are scattered throughout the chunk, it does not eliminate extent interleaving.

To display the location and size of the blocks of free space, execute the **onutil** CHECK SPACE command.

To use the CLUSTER keyword to try to eliminate extent interleaving

1. If the index that you want to cluster already exists, drop it.
2. Create the index that you want to cluster with the CLUSTER keyword of the CREATE INDEX statement.

   This step eliminates interleaving of the extents when you rebuild the table by rearranging the rows.

   The CREATE INDEX operation automatically rebuilds all other indexes on a table when it creates a clustered index. You compact the indexes in this step because the database server sorts the index values before it adds them to the B+ tree.
3. If you want to eliminate extent interleaving on other tables that reside in the chunk, repeat steps 1 and 2 for each table.

To prevent interleaved extents from recurring, consider increasing the size of the tblspace extents. For more information, see the *Informix Guide to SQL: Tutorial*.

Using ALTER TABLE to Eliminate Extent Interleaving

If you use the ALTER TABLE statement to add or drop a column or to change the data type of a column, the database server copies and reconstructs the table. When the database server reconstructs the entire table, the table is rewritten onto other extents in the dbspace. However, because other tables are still in the dbspace, the new extents might not be adjacent to each other.

Important: *If you only add one or more columns to the end of a table row, the database server does not copy and reconstruct the table during the ALTER TABLE operation. In this case, the database server uses an in-place alter algorithm to modify each row later, when it is updated. For more information on the conditions and syntax when this in-place ALTER TABLE algorithm is used, refer to the "Informix Guide to SQL: Syntax."*

Reclaiming Unused Space in an Extent

After the database server has allocated disk space to a tblspace as part of an extent, that space remains allocated to the tblspace. Even if all extent pages are empty after data is deleted, other tables cannot use their disk space.

Important: *When you delete rows in a table, the database server reuses that space to insert new rows in the same table. This section describes procedures to reclaim unused space for use by other tables.*

You can resize a table that does not require the entire amount of space that was originally allocated to it. Create a new table with smaller extent sizes, unload the data from the larger table, load it into the smaller table, and drop the original larger table. When you drop the original, larger table, you release the unneeded space for other tables to use.

To reclaim the disk space in empty extents and make it available to other tables, the database server administrator can rebuild the table. To rebuild the table, use any of the following methods:

- External tables

  If the table does not include an index, you can unload the table, re-create the table either in the same dbspace or in another one, and reload the data. Using external tables improves performance. For further information about using external tables to load and unload very large tables, refer to the *Administrator's Reference*.

- ALTER FRAGMENT INIT

  You can use the ALTER FRAGMENT INIT statement to rebuild a table, which releases space in the extents that were allocated to that table.

  For more information about the syntax of the ALTER FRAGMENT INIT statement, refer to the *Informix Guide to SQL: Syntax*.

- ALTER TABLE

  You can use the ALTER TABLE statement that invokes the slow alter algorithm to rebuild a table, which releases space in the extents that were allocated to that table.

  For information about using the ALTER TABLE statement to reclaim empty extent space, see"Altering a Table Definition" on page 6-34. For more information about the syntax of the ALTER TABLE statement, refer to the *Informix Guide to SQL: Syntax*.

Changing Tables

You might change an existing table for the following reasons, among others:

- To refresh large decision-support tables with data periodically
- To add or drop historical data from a specific time period
- To add, drop, or modify columns in large decision-support tables when the need arises for different data analysis

Loading and Unloading Tables

Databases for decision-support applications are often created by periodically loading and restructuring tables that have been unloaded from active OLTP databases. Data is often loaded with external tables in express mode into OPERATIONAL or RAW tables to get the advantage of light appends.

For example, you might use a series of SQL statements to restructure the data and store it in a temporary table. To take advantage of parallel processing and avoid logging overhead, use a SELECT.... INTO SCRATCH statement to unload table data into a temporary table in temporary dbspaces. Then create an external table and insert the temporary table into it. You then use the external table to load the data into your data mart or data warehouse.

For a complete description of unloading and loading data from external tables, see the *Administrator's Reference*.

If you expect to load and unload the same table often to build a data mart or data warehouse, monitor the progress of the job to estimate how long similar jobs will take in the future. To monitor a load job, first run **onstat -g sql to** obtain the session ID, and run **onstat -g xmp** to obtain the query ID for the session ID. Then you can run **onstat -g xqs** *query id* to get the runtime number of rows.

You can also enter the SET EXPLAIN ON statement to write the number of rows to be processed to the **sqlexplain.out** file and then use the **onstat** options to monitor the load process. Enter the SET PLOAD FILE statement to specify a file that stores statistics about the number of rows loaded and rejected and the location of the reject file.

Dropping Indexes Before You Load or Update Tables

In decision-support applications, you can confine most table updates to a single time period. You might be able to set up your system so that all updates are applied overnight or on specified dates.

When updates are performed as a batch, you can drop all nonunique indexes while you make updates and then create new indexes afterward. This strategy can have the following positive effects:

- The updating program can run faster with fewer indexes to update. Often, the total time to drop the indexes, update without them, and re-create them is less than the time to update with the indexes in place. For information about the time cost of updating indexes, see "Update-Time Costs" on page 7-12.
- Newly made indexes are more efficient. Frequent table updates tend to dilute the index structure so that it contains many partly full leaf pages. This dilution reduces the effectiveness of an index and wastes disk space.

To save time, make sure that a batch-updating program calls for rows in the sequence that the primary-key index defines. Then pages of the primary-key index are read in order and only once.

The presence of indexes also slows the population of tables when you use the LOAD statement or parallel inserts. Loading a table that has no indexes is fast, little more than a disk-to-disk sequential copy, but updating indexes adds a great deal of overhead.

To load a table that has no indexes

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the nonunique indexes.

To guarantee that the loaded data satisfies all unique constraints, create unique indexes and then load the rows in DELUXE mode, which modifies the index and checks constraints for each row as it is loaded. You save time if the rows are presented in the correct sequence for at least one of the indexes. If you have a choice, make it the row with the largest key. This strategy minimizes the number of leaf pages that must be read and written.

Using External Tables to Load and Unload Simple Large Objects

When you use external tables to load and unload data, you can define the table to specify simple-large-object columns as either raw or delimited, as described in the *Informix Guide to SQL: Syntax*.

The choice of raw or delimited format has the following performance implications:

- Raw format for simple large objects in external files does not have any conversion costs. In addition, because the simple-large-object length field is embedded in the row and the simple-large-object data follows the row in the data stream, simple-large-object data is read only once and then inserted into the existing field in the row where it belongs.
- Delimited format for simple large objects has specific conversion costs. The simple large objects in delimited format files are read and written at the position where the simple-large-object column is defined. This process requires more buffer space and might also require an extra read and write if buffer space is exceeded.

For more information about loading simple large objects from external tables, refer to the *Administrator's Reference*.

Attaching or Detaching Fragments

The ALTER FRAGMENT statement with its ATTACH and DETACH options is often used to perform data warehouse operations. ALTER FRAGMENT DETACH provides a way to delete a segment of the table data rapidly. Similarly, ALTER FRAGMENT ATTACH lets you load large amounts of data into an existing table incrementally. The attached table fragments must be the same type as the table to which they are to be attached, such as RAW or OPERATIONAL.

For more information about taking advantage of the performance enhancements for the ATTACH and DETACH options of the ALTER FRAGMENT statement, refer to "Attaching and Detaching Table Fragments" on page 9-62.

Altering a Table Definition

The database server chooses one of the following algorithms to process an ALTER TABLE statement in SQL:

- In-place alter
- Slow alter
- Fast alter

The algorithm chosen depends on the requested alteration, as described in "Alter Operations That Do Not Use the In-Place Alter Algorithm" on page 6-37.

In-Place ALTER TABLE

When you execute an ALTER TABLE statement that uses the in-place alter algorithm, the database server creates a new version of the table structure. When a row is updated, the in-place alter algorithm moves the altered row from the old definition of the table to the new definition. Rows are not duplicated in the old and new definition of the table.

The database server keeps track of all versions of table definitions. The database server retains the version status as well as all of the version structures and alter structures until the entire table is converted to the final format or a slow alter is performed. For complete information about ALTER TABLE syntax and restrictions, see the *Informix Guide to SQL: Syntax*.

The in-place alter algorithm provides the following time and disk-space performance advantages:

- It increases table availability and improves system throughput during the ALTER TABLE operation.

  The table is available for use sooner when the ALTER TABLE operation uses the in-place alter algorithm because the database server locks the table for only the time that it takes to update the table definition and rebuild indexes that contain altered columns.

  The table is locked for a shorter time than with the slow alter algorithm because the database server:

  - does not make a copy of the table in order to convert the table to the new definition or convert the data rows during the alter operation.
  - alters the physical rows in place with the latest definition after the alter operation when you subsequently update or insert rows. The database server converts the rows that reside on each page that you updated.
  - does not rebuild all indexes on the table.

  This increase in table availability can increase system throughput for application systems that require 24-by-7 operations.

- It requires less space than the slow alter algorithm because the database server:

  - does not make a copy of the table in order to convert the table to the new definition.
  - does not log any changes to the table data during the alter operation.

  These space savings can be substantial for very large tables.

Performance Considerations for DML Statements

If the database server detects a version page from a previous level during the execution of DML statements (INSERT, UPDATE, DELETE, SELECT), it performs the following actions:

- For UPDATE statements, the database server converts entire data pages to the final format.
- For INSERT statements, the database server converts the inserted row to the final format and inserts it in the page where it fits best. The database server converts the existing rows on the page to the final format.
- For DELETE statements, the database server does not convert the data pages to the final format.
- For SELECT statements, the database server does not convert the entire data page to the final format.

  If your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query because the database server reformats each row before it is returned.

Improving In-Place Alter Performance

As long as unconverted data pages exist, performance for updates and queries on the altered table might suffer because the database server must convert the data before processing it as requested.

An in-place ALTER TABLE is outstanding when data pages still exist with the old definition. The **onutil** CHECK TABLE command displays data page versions for outstanding in-place alter operations. The **Count** field displays the number of pages that currently use that version of the table definition.

To improve performance, you might convert any remaining unconverted data pages to the latest definition with a dummy UPDATE statement. For example, the following statement, which sets a column value to the existing value, causes the database server to convert data pages to the latest definition:

```
UPDATE tab1 SET col1 = col1;
```

After an update is executed on all pages of the table, execute the **onutil** CHECK TABLE command. The total number of data pages for the current version of the table appears in the **Count** field.

Important: *As you execute more ALTER TABLE statements that use the in-place alter algorithm on a table, each subsequent ALTER TABLE statement takes more time to execute than the previous statement. Therefore, Informix recommends that you not have more than approximately 50 to 60 outstanding alters on a table. Outstanding ALTER TABLE statements affect only the subsequent ALTER TABLE statements. They do not affect the performance of SELECT statements.*

Alter Operations That Do Not Use the In-Place Alter Algorithm

The database server does not use the in-place alter algorithm in the following situations:

- When you increase the length of a CHARACTER, DECIMAL, MONEY, or SMALLINT column and specify more than one algorithm

  If the ALTER TABLE statement contains more than one change, the database server uses the slower algorithm in the execution of the statement.

  For example, assume that an ALTER TABLE MODIFY statement extends a CHARACTER column and shrinks a DECIMAL column. Increasing the length of a CHARACTER column is an in-place alter operation that requires no data conversion, but decreasing the length of a DECIMAL column is a slow-alter operation because it might require data conversion. The database server uses the slow-alter algorithm to execute this statement.

- When you convert from real decimal to floating point

  Special considerations apply when you convert a real (fixed-point) decimal number to a floating-point number.

  A fixed-point DECIMAL column has the format DECIMAL(*p1*,*s1*), where *p1* refers to the precision of the column (the total number of significant digits) and *s1* refers to its scale (the number of digits to the right of the decimal point).

  If you are using an ANSI-mode database and specify DECIMAL(*p*), the value defaults to DECIMAL(*p*,0). In a non-ANSI database, the value is treated as a floating point with a precision of *p*.

  If a fixed-point DECIMAL is converted to a floating-point DECIMAL, a slow alter is performed.

In addition to these restrictions, the slow-alter algorithm is used instead of the in-place alter algorithm if:

- the modified column is used for hash partitioning in either a hash or hybrid fragmentation scheme.
- you drop or modify a column in a table that has a bitmap index.
- you drop or alter a simple-large-object column.

Altering a Column That Is Indexed

If an altered column is indexed, the table is still altered in place. The database server automatically rebuilds the index or indexes. If the index does not need to be rebuilt, improve performance by dropping or disabling the index before you perform the alter operation.

However, if the column that you modify is a primary key or foreign key and you want to keep this constraint, specify those keywords again in the ALTER TABLE statement. The database server then rebuilds the index.

For example, suppose that you create tables and alter the parent table with the following SQL statements:

```
CREATE TABLE parent
    (si smallint PRIMARY KEY CONSTRAINT pkey);
CREATE TABLE child
    (si smallint REFERENCES parent ON DELETE CASCADE
    CONSTRAINT ckey);
INSERT INTO parent (si) VALUES (1);
INSERT INTO parent (si) VALUES (2);
INSERT INTO child (si) VALUES (1);
INSERT INTO child (si) VALUES (2);

ALTER TABLE parent
     MODIFY (si int PRIMARY KEY CONSTRAINT pkey);
```

This ALTER TABLE example converts a SMALLINT column to an INT column.The database server retains the primary key because the ALTER TABLE statement specifies the PRIMARY KEY keywords and the **pkey** constraint. However, the database server drops any referential constraints that reference that primary key. Therefore, you must also specify the following ALTER TABLE statement for the child table:

```
ALTER TABLE child
    MODIFY (si int REFERENCES parent ON DELETE CASCADE
            CONSTRAINT ckey);
```

Even though the ALTER TABLE operation on a primary key or foreign key column rebuilds the index, the database server still takes advantage of the in-place alter algorithm to provide the following performance benefits:

- Does not make a copy of the table in order to convert the table to the new definition
- Does not convert the data rows during the alter operation
- Does not rebuild all indexes on the table

Warning: *If you alter a table that is part of a view, you must re-create the view to obtain the latest definition of the table.*

Slow ALTER TABLE

When the database server uses the slow alter algorithm, the table is locked for a long period of time because the database server:

- makes a copy of the table in order to convert the table to the new definition.
- converts the data rows during the alter operation.

The database server uses the slow alter algorithm when the ALTER TABLE statement changes columns that cannot be changed in place. For more information, see "Alter Operations That Do Not Use the In-Place Alter Algorithm" on page 6-37.

Fast ALTER TABLE

The database server uses the fast alter algorithm when the ALTER TABLE statement changes attributes of the table and does not affect the data. The database server uses the fast alter algorithm when the ALTER TABLE statement changes the following attributes:

- The type of table (RAW, STATIC, OPERATIONAL, and STANDARD)
- The lock mode of the table
- The next-extent size
- Constraints

When the database server uses the fast alter algorithm, the table is locked for a short time. In some cases, the database server locks the system catalog tables to change the attribute. In either case, the table is unavailable for queries only briefly.

Denormalizing the Data Model to Improve Performance

Operational databases for OLTP transactions are usually constructed with the entity-relationship data model described in the *Informix Guide to SQL: Tutorial*. This model produces tables that contain no redundant or derived data and ensures data integrity. These tables are well structured by the tenets of relational theory.

Databases for data warehouses and data marts are usually constructed with a different model, sometimes referred to as a star schema or fact-dimension table model. This model is still relational in part, but it is constructed with different goals in mind. A *fact-dimension* database contains one large fact table and several smaller dimension tables. Each dimension table corresponds to a key in the fact table and contains descriptive information.

For operational OLTP databases, the most frequent denormalization techniques are as follows:

- Creating companion tables that contain columns that are used less often
- Modifying table definitions to create shorter rows
- Building symbol tables
- Adding redundant data, either columns or tables

Fact-dimension database schemas for data warehouse applications usually contain redundant data in the dimension tables for efficiency. Some of the techniques described in "Adding Redundant Data" on page 6-45, such as replicating tables across coservers, can also be used to improve performance of data warehouse queries.

Important: *If the database designer modifies the data model to meet special demands for high performance, changes made to the database and table structure might also require adjustments by the database application engineers.*

Creating Companion Tables

Some of the methods described in the following sections involve splitting tables to create companion tables. Evaluate your applications and database use carefully before you denormalize the database in this way.

Creating companion tables has the following three disadvantages:

- Each table consumes extra disk space and adds complexity.
- Two copies of the primary key occur for each row, one copy in each table. Two primary-key indexes also exist.

  To estimate the number of added pages, you can use the methods described in "Estimating Table Size" on page 6-15.
- You must modify existing programs, reports, and forms that use SELECT * because fewer columns are returned. Programs, reports, and forms that use attributes from both tables must perform a join to bring the tables together.

  If many queries require table joins, the performance degradation might be unacceptable.

  In this case, when you insert or delete a row, two tables are altered instead of one. If you do not coordinate the alteration of the two tables (by making them in a single transaction, for example), you lose semantic integrity.

Using Shorter Rows for Faster Queries

Tables with shorter rows yield better performance than ones with longer rows because disk I/O is performed in pages, not in rows. The shorter the rows of a table, the more rows occur on a page. The more rows per page, the fewer I/O operations it takes to read the table sequentially and the more likely it is that nonsequential access can be performed from data already in a buffer. Shorter rows are also transferred faster between coservers.

The entity-relationship data model puts all the attributes of one entity into a single table for that entity. For some entities, this strategy can produce very long rows. To shorten the rows, you might create companion tables by separating columns into tables with duplicate key values. As the rows get shorter, query performance should improve, but only if the companion tables are not joined in most queries.

Expelling Long Strings

The largest attributes are often character strings. Removing them from the entity table makes the rows shorter. You can use the following methods to expel long strings. The first two methods are relatively simple; the other two methods require significant changes in application programs:

- Use VARCHAR columns.
- Use TEXT columns.
- Move strings to a companion table.

You can also build a symbol table.

Using VARCHAR Columns

GLS

A database might contain CHAR columns that can be converted to VARCHAR columns. You can use a VARCHAR column to shorten the average row length when the average length of the text string in the CHAR column is at least 2 bytes shorter than the width of the column. For information about the advantages of different character data types, refer to the *Informix Guide to GLS Functionality*. ♦

VARCHAR data is immediately compatible with most existing programs, forms, and reports. You might need to recompile any forms produced by application development tools to recognize VARCHAR columns. Always test forms and reports on a sample database after you modify the table schema.

Using TEXT Columns

When a string fills half a disk page or more, consider converting it to a TEXT column, which is stored in a separate page in the dbspace with the table.The length of the column in the data page is a pointer only 56 bytes long to the TEXT data page. However, the TEXT data type is not automatically compatible with existing programs. The code needed to fetch a TEXT value is more complicated than the code to fetch a CHAR value into a program.

Moving Strings to a Companion Table

Strings shorter than half a page waste disk space if you treat them as TEXT columns, but you can move them from the main table to a companion table.

Important: *If you move strings to a companion table, queries and other application program entities must be changed to join the tables. If too many changes must be made or too many joins must occur, this method might not improve performance. The same warning applies to the method described in the next section,* *"Building a Symbol Table."*

Building a Symbol Table

If a column contains strings that are not unique in each row, you can move those strings to a table in which only unique copies are stored.

For example, in a CUSTOMER table, the **customer.city** column contains city names. Some city names are repeated down the column, and most rows have some trailing blanks in the field. Using the VARCHAR data type eliminates the blanks but not the duplication.

You can create a table named **cities**, as the following example shows:

```
CREATE TABLE cities (
    city_num SERIAL PRIMARY KEY,
    city_name VARCHAR(40) UNIQUE
)
```

You can change the definition of the **customer** table so that its **city** column becomes a foreign key that references the **city_num** column in the **cities** table.

You must change any program that inserts a new row into **customer** to insert the city of the new customer into **cities**. The database server return code in the **SQLCODE** field of the SQL Communications Area (SQLCA) can indicate that the insert failed because of a duplicate key. If this error occurs, it simply means that an existing customer is located in that city. For information about the SQLCA, refer to the *Informix Guide to SQL: Tutorial*.

In addition to changing programs that insert data, you must also change all programs and stored queries that retrieve the city name. The programs and stored queries must use a join into the new **cities** table to obtain their data. The extra complexity in programs that insert rows and the extra complexity in some queries is the result of giving up theoretical correctness in the data model. Before you make the change, be sure that it returns a reasonable savings in disk space or execution time.

Splitting Wide Tables

Consider all the attributes of an entity that has rows that are too wide for good performance. Look for some theme or principle to divide them into two groups, and examine stored queries to find out what columns are used most often. Split the table into two tables, a primary table and a companion table, repeating the primary key in each one. The shorter rows allow you to query or update each table quickly.

Before you split a wide table, however, carefully evaluate the performance cost, as described in "Creating Companion Tables" on page 6-41.

Dividing by Bulk

One principle on which you can divide an entity table is bulk. Move the bulky attributes, which are usually character strings, to the companion table. Keep the numeric and other small attributes in the primary table. In the **stores_demo** demonstration database used for the *Informix Guide to SQL: Tutorial*, you can split the **ship_instruct** column from the **orders** table. You can call the companion table **orders_ship**. It has two columns, a primary key that is a copy of **orders.order_num** and the original **ship_instruct** column.

Dividing by Frequency of Use

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, you can move them to a companion table. For example, in the demonstration database, you query the **ship_instruct**, **ship_weight**, and **ship_charge** columns in only one program. You can move these columns to a companion table.

Dividing by Frequency of Update

Updates take longer than queries, and updating programs lock index pages and rows of data during the update process, which prevents querying programs from accessing the tables. If you can separate one table into two companion tables, one of which contains the most updated entities and the other of which contains the most queried entities, you can often improve overall response time.

Adding Redundant Data

Normalized databases contain no redundant tables or data. Every attribute appears in only one table. Normalized tables also contain no derived data. Instead, data that can be computed from existing attributes is selected as an expression based on those attributes.

Normalizing tables minimizes the amount of disk space used and expedites updating tables. However, in databases used for data warehouses and data marts, derived and redundant data can improve query processing time by reducing the necessity of joining tables and performing aggregations.

As an alternative, you can introduce new columns that contain redundant data and duplicate small tables on each coserver, provided that you understand the trade-offs involved. You can also create special indexes that contain summary, selected, and prejoined row data, as well as composite keys, as described in "Using Indexes" on page 13-13.

Adding Redundant Data to Tables

A correct data model avoids redundancy by keeping attributes only in the table for the entity that they describe. If the attribute data is needed in a different context, you make the connection by joining tables. But joining takes time. If a frequently used join affects performance, you can eliminate it by duplicating the joined data in another table.

The disadvantage of adding redundant data to tables is that it takes space and poses an integrity risk. If the data is updated in one table, it must also be updated in other tables where it is duplicated. For more information, refer to the *Informix Guide to SQL: Syntax* and the *Informix Guide to SQL: Reference*.

Adding Redundant Tables

Duplicating some small tables across all coservers can improve performance for both OLTP and DSS applications. In DSS queries, the database server might automatically replicate a table smaller than 128 kilobytes that is involved in a hash join if such replication would improve performance. This feature is called *small-table broadcast*. For more information, refer to "Balanced Workload" on page 11-13.

Keeping Small Tables in Memory

Tables or indexes or fragments of tables or indexes that are in constant use and are smaller than 10 or 12 megabytes can be made memory resident to increase lookup efficiency. For fragmented tables or indexes, you can specify residency for individual fragments. Memory-resident tables are cached in the buffer pool of the coserver where the table, index, or fragment exists.

To specify that a table be cached in the buffer pool, use the SET Residency statement, as shown in the following examples:

```
SET TABLE tab1 MEMORY_RESIDENT
```

A memory-resident table or index remains in the buffer pool until one of the following events occurs:

- You use the SET Residency statement to set the database object to NON_RESIDENT.
- The database object is dropped.
- The database server is brought down.

Each time the database server is started you must specify the tables and indexes that you want to cache in the buffer pool.

If you cache tables in the buffer pool, you should monitor memory buffers carefully to make sure that they are being used efficiently and that table or index fragments resident in the buffer pool do not cause foreground writes to occur. To monitor memory buffers and memory-resident tables and display the number of resident buffers for each partition, use **onstat -P**. To list all memory buffers, use **onstat -B**.

For complete information about using the SET Residency statement, refer to the *Informix Guide to SQL: Syntax*.

Chapter

7

Index Performance

In This Chapter

This chapter describes general performance considerations associated with indexes. It discusses the following issues:

- Selecting an appropriate index type
- Estimating the size of indexes
- Managing indexes for disk-space efficiency and query performance
- Improving the performance of index builds

For information about the performance advantages of specific index types, including examples, see Chapter 13, "Improving Query and Transaction Performance."

Choosing Index Types

Extended Parallel Server provides several unique kinds of indexes. These indexes improve performance when they are used appropriately. This section provides a summary of the index types and their general use.

To decide what kinds of indexes will improve query and transaction performance examine the queries and transactions run on your system and evaluate the database structure. For information about when a particular kind of index can improve query performance, see "Using Indexes" on page 13-13.

Conventional indexes are stored as B-tree indexes. For information about the structure of a B-tree index, see "Structure of a B-Tree Index" on page 7-4.

A *bitmap* index is a specialized variation of a B-tree index that is useful for indexing columns that can contain one of only a few values, such as marital status or gender. For each highly duplicate value, a bitmap index stores a compressed bitmap for each value that the column might contain. Each bit in the bitmap represents one row in the table. A bit in the bitmap is turned on if the table row that it represents contains the value that this bitmap indexes.

Generalized Key Indexes

Generalized key (GK) indexes let you store the result of an expression as a key. The three kinds of GK indexes are selective, virtual column, and join.

Because GK indexes are permitted only on STATIC tables, they are most useful in DSS applications that use stable data and in OLTP applications that use lookup tables that do not change.

For detailed information about these index types, when and how to create them, and when the optimizer uses them, see "Using Generalized-Key Indexes" on page 13-23.

Structure of a B-Tree Index

This section explains how a B-tree index is organized. Information about index structure can help you understand how indexes are used and why re-creating an index might improve index efficiency.

As Figure 7-1 shows, a B-tree index is arranged as a hierarchy of pages, which is technically a *B+ tree*. The top level of the hierarchy contains a single *root page*. Intermediate levels, when needed, contain *branch pages*. Each branch page contains entries that refer to a subset of pages in the next level of the index. The bottom level of the index contains a set of *leaf page*s. Each leaf page contains a list of index entries that refer to rows in the table.

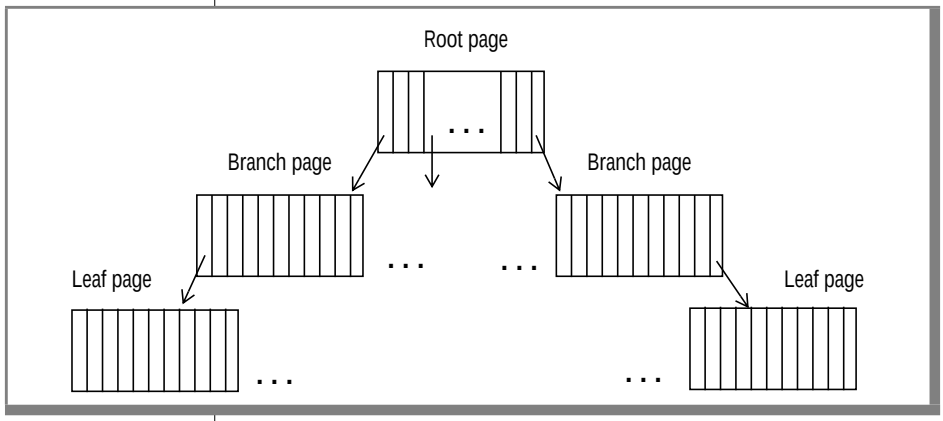


Figure 7-1
B-Tree Structure of an Index

The number of levels needed to hold an index depends on the number of unique keys in the index and the number of index entries that each page can hold. The number of entries per page depends, in turn, on the size of the columns being indexed.

For example, if the index page for a given table can hold 100 keys, a table of up to 100 rows requires a single index level: the root page. When this table grows beyond 100 rows, to a size between 101 and 10,000 rows, it requires a two-level index: a root page and between 2 and 100 leaf pages. When the table grows beyond 10,000 rows, to a size between 10,001 and 1,000,000 rows, it requires a three-level index: the root page, a set of 100 branch pages, and a set of up to 10,000 leaf pages.

For more information about the structure of B-tree indexes, refer to the *Administrator's Reference*.

Estimating Index Page Size

The index pages associated with a table can add significantly to the size of a dbspace. An attached index for a table is stored in the same dbspace as the table, but in a different tblspace. A detached index is stored in different dbspaces that you specify.

The database sever determines the index extent size, based on the extent size of the table and the index key length and the table row size. The formula for estimating the index extent size is as follows:

```
Index extent size = table extent size * ((key length + 8)/row size)
```

The minimum extent size is four pages.

Estimating Conventional Index Page Size

For information about bitmap index size estimates, see "Estimating Bitmap Index Size" on page 7-8.

To estimate the number of index pages

1. Add up the total widths of the indexed column or columns. This value is referred to as *colsize*. For a nonfragmented index, add 5 to *colsize* to obtain *keysize*, the actual size of a key in the index. For a fragmented index, add 9 to *colsize*.
2. Calculate the expected proportion of unique entries to the total number of rows. This value is referred to as *propunique*. If the index is unique or rows contain very few duplicate values, use `1` for *propunique*. If a significant proportion of entries are duplicates, divide the number of unique index entries by the number of rows in the table to obtain a fractional value for *propunique*. If the resulting value for *propunique* is less than `.01`, use `.01` in the calculations that follow.

3. Estimate the size of a typical index entry with one of the following formulas, depending on whether the table is fragmented or not:

   a. For nonfragmented tables, use the following formula:

   ```
   entrysize = keysize * propunique + 5
   ```

   b. For fragmented tables, use the following formula:

   ```
   entrysize = keysize * propunique + 9
   ```

4. Estimate the number of entries per index page with the following formula:

   ```
   pagents = trunc(pagefree/entrysize)
   ```

   pagefree is the page size minus the page header (24 bytes).

   The **trunc()** function notation indicates that you should round down to the nearest integer value.

5. Estimate the number of leaf pages with the following formula:

   ```
   leaves = ceiling(rows/pagents)
   ```

   The **ceiling()** function notation indicates that you should round up to the nearest integer value; *rows* is the number of rows that you expect to be in the table.

6. Estimate the number of branch pages at the second level of the index with the following formula:

   $branches_0 = ceiling(leaves/pagents)$

7. If the value of $branches_0$ is greater than 1, more levels remain in the index. To calculate the number of pages contained in the next level of the index, use the following formula:

   $branches_{n+1} = ceiling(branches_n/pagents)$

   $branches_n$ is the number of branches for the last index level that you calculated.

   $branches_{n+1}$ is the number of branches in the next level.

8. Repeat the calculation in step 7 for each level of the index until the value of $branches_{n+1}$ equals 1.

9. Add up the total number of pages for all branch levels calculated in steps 6 through 8. This sum is referred to as *branchtotal*.

10. Use the following formula to calculate the number of pages in the compact index:

    ```
    compactpages = (leaves + branchtotal)
    ```

11. To incorporate the fill factor into your estimate for index pages, use the following formula:

    ```
    indexpages = 100 * compactpages / FILLFACTOR
    ```

    The default value of FILLFACTOR is 90. To decrease the size of the index and make index pages compact, specify a higher FILLFACTOR. To allow room for expansion in each index page but increase the size of the index, specify a lower FILLFACTOR.

As rows are deleted and new ones are inserted, the number of index entries in a page varies. This method for estimating index pages yields a conservative (high) estimate for most indexes. For a more precise value, build a large test index with real data and check its size with the **onutil** CHECK INDEX utility.

Estimating Bitmap Index Size

You can create bitmap versions of B-tree indexes, which can be used for tables that are updated, and of GK indexes, which can be used only for STATIC tables, which are not updated.

Conventional B-tree index and B-tree indexes with compressed bitmap leaf pages differ only in the way in which the duplicate list for each key is stored at the leaf level. In a B-tree index, duplicate values are stored in the same way as unique values, one in each slot of an index leaf page. In a bitmap index, duplicate values are stored as a bit map in a leaf page, with a bit mapped to each row of the table. For rows that contain the specific value, the corresponding bit is marked ON.

You might create a bitmap index if the column to be indexed contains many identical values. The more identical values that the column contains, the more efficient the index is, both in storage and processing. The more accurately you can estimate the number of identical values in the key columns, the more accurate your estimate of the index size and storage efficiency will be.

For example, if the indexed column can contain only two values, such as M and F, calculating the storage efficiency of a bitmap index is easy. On the other hand, if a column can contain fifty or sixty values, such as the ages of employees, calculating the efficiency of the index is harder, but a bitmap index on this column might still store key values more efficiently than a conventional B-tree index.

Follow these steps to determine the efficiency of a bitmap index on a table column and to estimate the disk space that the index will require:

1. Use the maximum number of table rows that can fit in a table page of 255 slots to calculate the integer *i*, which is the first power of 2 that is greater than or equal to the number of rows on a data page.

   For example, if a data page can contain a maximum of 35 rows, *i* is 6, because $2^5 < 35 <= 2^6$. In fact, the value of *i* is 6 for any number of rows between 33 and 64.

2. Estimate the average distance between two rows that have keys with the same value, assuming that keys with identical values are distributed evenly in the table fragment:

   - If the table is not fragmented, list the rowids with the following SELECT statement (replacing *key* with the column on which the index will be added and *duplicate_value* with the value of the duplicate key):

     ```
     SELECT rowid FROM table WHERE key = duplicate_value
     ```

   - If the table is fragmented, use the following **onutil** CHECK INDEX command to list all rowids and their associated keys:

     ```
     onutil
     >check index with data database database_name
        index index_name display data
     ```

   - The last byte of the hexadecimal row identifier is the slot number. The first three bytes (leading zeros suppressed) are the page ID. Use the following formula to calculate the distance between any two specific row identifiers:

     $(PageID_2 - PageID_1) * 2^i + (slot\#_2 - slot\#_1)$

     For example, if each page contains 35 rows, the distance between row identifier 0x302 and row identifier 0x401 is calculated as `(4 - 3) * 64 + (2 - 1) = 65`.

3. To calculate the integer, *N*, for a data fragment that contains a total number of pages, *P*, use the following formula:

   $N = P * 2^i$

   For example, if *P* is 10,000 and *i* is 6, then *N* is 640,000.

4. Given these formulas, use your estimate of the number of rows in the table and the number of duplicate key values to estimate the minimum space that is required for each bitmap at the leaf nodes of the index:

   - If the average row identifier distance between duplicate-key records, as calculated by the formula in step 2, is greater than 64, use the following formula to estimate the number of bytes required for the bitmap for a particular key value:

     ```
     bitmap_size = 24 + (8 * Number of duplicate-key
     records)
     ```

   - If the average row identifier distance between duplicate-key records is less than or equal to 64, use the following formula to estimate the space required for each compressed bitmap key:

     ```
     bitmap_size = 28 + N/8 bytes
     ```

Storage efficiency increases as the distance between rows with the same key *decreases*. For example, if the average row identifier distance is 75, the space required for a key with 100 duplicates is 824 bytes.

```
bitmap_size = 24 + (8 * 100) = 824 bytes
```

In this example, the 824 bytes to store the bitmap is more than the space needed for the key in a conventional index (5 * 100 = approximately 500 bytes plus the size of the key value).

However, if the value of *N* is `640,000`, and the row identifier distance between duplicate key records is 40, the space required for a key is 80,028 bytes.

```
bitmap_size = 28 + 640,000/8 = 80,028 bytes
```

In this case, the bitmap is much more efficient than the 3,200,000 bytes needed to store the duplicate row identifiers in a conventional index.

If the bitmap would be larger than the row identifier list of a conventional B-tree index, the database server uses the row identifier list representation for the key instead of the bitmap representation.

As with conventional B-tree indexes, only one bitmap value is permitted on an overflow page.

In addition, bitmaps are aligned on the leaf nodes on 4-byte boundaries. On each page, gaps of 0 to 3 bytes might exist between the end of the key and the beginning of its bitmap. Consider the previous example, where *N* is `640,000`, the row identifier distance between duplicate-key records is 40, and the space that is required for the bitmap is 80,028 bytes. If each index leaf page contains 4,000 free bytes and if the key size is 5 bytes, the bitmap is broken up into 3992- byte pieces because the space required for the key is rounded up to 8 bytes.

Managing Indexes

Indexes allow the database server to improve query and transaction processing. The optimizer can choose to use an index to improve performance in the following ways:

- To provide the optimizer with the possibility of an index join instead of a table join
- To avoid sequential scans for low selectivity searches
- To avoid reading row data when the database server processes expressions that name only indexed columns
- To avoid a sort (including building a temporary table) when the database server executes the GROUP BY and ORDER BY clauses
- To use only the index to perform aggregate operations

Indexes must be designed for your data and queries, however. For descriptions of index types and information about when they are useful, see "Using Indexes" on page 13-13.

Tip: *By default, the lock granularity for the index matches the lock granularity of the table, which might require many locks for index accesses. To reduce lock overhead for indexes on tables for which the key values do not change, you might change the lock granularity, as described in "Setting COARSE Locking for Indexes" on page 8-8.*

You can create several indexes on a single table. In some cases, the query optimizer can choose to use more than one index in query processing. Primarily, however, multiple indexes on a table allow the optimizer to choose an index that provides a more efficient query plan.

Evaluating Index Costs

Although indexes can speed query execution, you should consider the cost of their disk space and update time.

Disk-Space Costs

One cost of an index is disk space. Estimating methods for different indexes appears in "Estimating Conventional Index Page Size" on page 7-6 and "Estimating Bitmap Index Size" on page 7-8. An index can require large amounts of disk space. For an indexed table, you might have as many index pages as data pages.

Update-Time Costs

For performance tuning, a more important cost of an index is update time whenever the table is modified. If you index tables that are modified at specific times by load jobs, such as tables in databases used exclusively for DSS queries, the cost of updating the index is incurred at load time. The cost of updating an index might significantly affect performance of OLTP applications that update tables continuously.

The following descriptions of update-time cost assume that approximately two pages must be read to locate an index entry, as is the case when the index consists of a root page, one level of branch pages, and a set of leaf pages. The root page is assumed to be in a buffer already. The index for a very large table has at least two intermediate levels, so about three pages are read when you reference such an index.

A second assumption is that one index is used to locate a row that is being altered. The pages for that index might be found in page buffers in shared memory. However, the pages for any other indexes that need altering must be read from disk. Presumably, one index is used to locate a row being altered.

Under these assumptions, conventional B-tree index maintenance adds different amounts of time for different kinds of modifications, as the following list describes:

- When you delete a row from a table, its entries must be deleted from all indexes.

  You must look up the entry for the deleted row (two or three pages read in), and you must rewrite the leaf page. The write operation to update the index is performed in memory, and the leaf page is flushed when the least-recently-used (LRU) buffer that contains the modified page is cleaned. So this operation requires two or three page accesses to read the index pages if needed, and one deferred page access to write the modified page.

- When you insert a row, its entries must be inserted in all indexes.

  A place in which to enter the inserted row must be found within each index (two or three pages read in) and rewritten (one deferred page out), for a total of three or four immediate page accesses per index.

  When you update a row, its entries must be looked up in each index that applies to an altered column (two or three pages in).

  The leaf page must be rewritten to eliminate the old entry (one deferred page out), the new column value must be located in the same index (two or three more pages in), and the row must be entered (one more deferred page out).

Insertions and deletions change the number of entries on a leaf page. Almost every *pagents* operation requires some additional work to deal with a leaf page that has either filled up or been emptied. However, if *pagents* is greater than 100, this additional work occurs less than 1 percent of the time. You can often ignore this work when you estimate the I/O impact. (The calculation and definition for *pagents* appears in "Estimating Conventional Index Page Size" on page 7-6.)

In short, when a row is inserted or deleted at random, three to four added page I/O operations occur for each index. When a row is updated, allow six to eight page I/O operations for each index that applies to an altered column.

If a transaction is rolled back, all this work must be undone. For this reason, rolling back a transaction can take a long time.

Because the alteration of the row itself requires only two page I/O operations, index maintenance can be the most time-consuming part of data modification. For information about one way to reduce this cost, see "Dropping Indexes" on page 7-19.

Choosing an Attached or Detached Index

The performance advantages of an attached or detached index depend on how queries and transactions use the index. The optimizer considers both attached and detached indexes for a multi-index plan. The optimizer cost estimates might result in using a combination of attached and detached indexes for a multi-index scan.

An attached index is fragmented in the same way as the underlying table, with the same fragmentation scheme, and is stored in the same dbspaces but in different tblspaces. If the index will be used to access the table fragment that is stored in the same dbspace, create an attached index.

In general, attached indexes are considerably more efficient than detached indexes, especially globally detached indexes. Nevertheless, you might create detached indexes to provide unique constraints on keys that are not used to fragment the table.

A detached index is either fragmented differently from the underlying table, or nonfragmented when the table is fragmented. It can be stored in any specified dbspaces or dbslices. Indexes can be fragmented with any fragmentation scheme except round-robin.

Locally detached index fragments are stored on the same coserver as the corresponding table fragments. Locally detached indexes are faster than globally detached indexes, even for SELECT statements, because they do not incur high messaging costs. Maintenance costs for locally detached indexes are also lower than for globally detached indexes.

Globally detached index fragments are stored on coservers differently from the table fragments because the index is fragmented differently from the table. Globally detached indexes also allow unique constraints on any column.

Globally detached indexes increase maintenance costs for deletes, inserts, and updates of the index key column data. If the underlying table rarely changes, these costs should be insignificant, but the intercoserver messaging cost of using a globally detached index is high.

A globally detached index might take advantage of fragment elimination or reduced disk I/O to improve performance of queries with the following requirements:

- Selection of a single row based on columns that are not used for table fragmentation
- A key-only scan of the columns only in the detached index fragment, as specified by the WHERE clause
- A join that involves either selection of a single row or a key-only scan of only the detached index fragment

In all other cases, a nonfragmented index, an attached index, or a locally detached index provides better performance.

Important: *You cannot create a FOR EACH ROW trigger on a table that has a globally detached index.*

Setting the Lock Mode for Indexes

By default, indexes inherit the lock granularity of the indexed table. If the lock granularity of the indexed table is row locking, the lock mode of the index is also row locking. Row or page locking adds unnecessary lock overhead to index accesses for indexes in which the keys are rarely or never changed.

If you know that index-key data will not change, you can improve performance by setting the index lock mode to COARSE with the ALTER INDEX statement, as explained in "Setting COARSE Locking for Indexes" on page 8-8.

In COARSE lock mode, the database server places a shared lock on the entire index fragment instead of using the key or page-level locking specified when the index was created.

Choosing Columns for Indexes

The columns that you choose for an index depend on the data in the columns and the queries that might use the index:

- Add a conventional index on columns that:
  - are used in joins.
  - are frequently used in filter expressions.
  - are frequently used for ordering or grouping.
  - are used for aggregation
  - are used in selection
- Add a bitmap index on columns that contain duplicate keys, especially if many columns contain the same key value, such as a gender or marital-status identifier.

  For information about when to use bitmap indexes, see "Using Bitmap Indexes" on page 13-20.
- Add a GK index for a STATIC table on virtual columns, selected columns, or columns joined from other STATIC tables, as suggested in "Using Generalized-Key Indexes" on page 13-23.

Indexing Filter Columns in Large Tables

If a column in a large table is often used as a filter in a WHERE clause, consider placing an index on it. The optimizer can use the index to select the required rows and avoid a sequential scan of the entire table. One example is a table that contains a large mailing list. If you find that a postal-code column is often used to filter data, consider creating an index on that column.

This strategy yields a net savings of time only when the cardinality of the column is high; that is, when indexed values are duplicated in only a small fraction of rows. Nonsequential access through an index takes more disk I/O operations than does sequential access. Therefore, if a filter expression on the column passes more than a quarter of the rows, the database server might as well read the table sequentially. As a rule, indexing a filter column saves time in the following cases:

- The column is used in filter expressions in many queries or in slow queries.
- Most column values appear in fewer than 10 percent of the rows, which indicates a selectivity of more than 90 percent.

Indexing Order-By and Group-By Columns

When many rows must be ordered or grouped, the database server must put the rows in order. One way that the database server performs this task is to select all the rows into a temporary table and sort the table. However, if the ordering columns are indexed, the optimizer can sometimes read the rows in sorted order through the index and avoid a final sort. For information about how indexes improve query performance, see "Using Indexes" on page 13-13.

Because the keys in an index are in sorted sequence, the index actually represents the result of sorting the table. When you place an index on the ordering column or columns, if the index keys are sorted in the order required by the query, you can replace many sorts during queries with a single sort that occurs when the index is created.

Avoiding Columns with Duplicate Keys

When duplicate keys are indexed in a conventional B-tree index, entries that match a given key value are grouped in lists. The database server uses these lists to locate rows that match a requested key value.

When the selectivity of the index column is high, these lists are generally short. But when only a few unique values occur, the lists become long and can even cross multiple leaf pages.

Creating a conventional index on a column that has low selectivity (that is, a small number of distinct values relative to the number of rows) can actually slow performance because of the cost of searching through the duplicate index key values for the rows that satisfy the query.

You can address this problem in both of the following ways:

- If the data column that you want to index has a low selectivity, create the index with the USING BITMAP expression. To find out if a bitmap index will be efficient, see "Estimating Bitmap Index Size" on page 7-8.

  For information about the CREATE INDEX statement with the USING BITMAP expression, refer to the *Informix Guide to SQL: Syntax*.

- You might also replace the index on the low-selectivity column with a composite index that has higher selectivity. Use the low-selectivity column as the leading column and a high-selectivity column as the second column in the index.

  For more information about when and how to create composite indexes, refer to "Using Composite Indexes" on page 13-21.

Clustering Indexes

Clustering indexes can be used for the following purposes:

- To prevent table-extent interleaving
- To reduce nonsequential access costs

You can have only one clustering index on a table although additional indexes on other columns in the table are permitted.

B-tree clustering indexes are often created on columns that do not have unique values for each row, such as a postal-code column. The table itself is physically organized in blocks in such a way that the clustering index contains a pointer to the first table block that contains a given value for the clustering column.

Important: *You cannot create clustering indexes on STANDARD tables in Extended Parallel Server. If you need a clustering index on a table, first convert the table to OPERATIONAL or STATIC type. Then create the clustering index, perform a level-0 backup of the dbspaces involved, and convert the table back to STANDARD type. The clustering order is not retained when the table is updated.*

You can be sure that when the table is searched on the indexed column in a clustering index, it is read in sequential instead of nonsequential order. For a discussion of these issues, see Chapter 10, "Queries and the Query Optimizer."

Clustering is not preserved when you make changes to a table. When you insert new rows into a table organized as a B-tree clustering index, the rows are stored physically at the end of the table regardless of their clustering key value. When you update rows and change the value of the clustering column, the rows are written back into their original location in the table.

Clustering and reclustering consume a large amount of space and time. You can avoid some of these costs if you build or load the table in the desired order.

Dropping Indexes

When an update transaction commits, the database server btree cleaner removes deleted index entries and, if necessary, rebalances the index nodes. However, depending on the order in which your application adds and deletes keys from the index, the structure of an index can become inefficient.

Use the **onutil** CHECK TABLE INFO command with the TABLESPACE option, as described in the *Administrator's Reference*, to determine the amount of free space in each index page. If your table has relatively low update activity and a large amount of free space exists, you might drop and re-create the index with a higher value for FILLFACTOR to make the unused disk space available for other uses.

Dropping Indexes Before Table Updates

In some applications, most table updates can be confined to a single time period. You might be able to set up your system so that all updates are applied overnight or on specified dates.

When updates are performed as a batch, drop all nonunique indexes before you update tables and then create new indexes afterward. This strategy can have the following positive effects:

- The updating program runs faster with fewer indexes to update. Often, if the number of updates is large, the total time to drop the indexes, update without them, and re-create them is less than the time to update with the indexes in place. For the time cost of updating indexes, see "Update-Time Costs" on page 7-12.
- Newly made indexes are more efficient. Frequent updates tend to dilute the index structure so that it contains many partly full leaf pages. This dilution reduces the effectiveness of an index and wastes disk space.

As a time-saving measure, make sure that a batch-updating program calls for rows in the sequence defined by the primary-key index. That sequence causes the pages of the primary-key index to be read in order and only once.

The presence of indexes might also slow the population of tables when you use the LOAD statement. Loading a table that has no indexes is quick because it is little more than a disk-to-disk sequential copy.

The amount of data that you are loading determines whether it is more efficient to drop an index on a table. If you are loading a small amount of data, it might be efficient to retain the index and use a load method that updates the indexes. If you are loading a large amount of data, it might be efficient to drop all indexes first and rebuild them after you load the new data.

For information about the steps for loading tables after you drop indexes, see"Loading and Unloading Tables" on page 6-31.

Maintaining Index Space Efficiency

When an update transaction commits, the database server btree cleaner removes deleted index entries and balances the index nodes. However, depending on the order in which your application adds and deletes keys from the index, the structure of an index might become inefficient. Frequent updates tend to expand the index structure, so that it contains many partly full leaf pages. This expansion:

- reduces the effectiveness of an index because more I/O operations might be needed to scan the index.
- wastes disk space.

Use the **onutil** CHECK TABLE command with the TABLESPACE and ALLOCATION INFO options to find out how much free space is in each index page. If the table has relatively little update activity and contains a large amount of free space, drop and re-create the index with a larger value for FILLFACTOR to make the unused disk space available for other uses.

You can also use the **onutil** CHECK INDEX KEYS command to verify the integrity and consistency of B-tree indexes.

For the following information, refer to the *Administrator's Reference*:

- How the database server maintains an index tree
- How to use **onutil**

Increasing Concurrency During Index Checks

To verify that indexes are correctly constructed and contain valid entries, use the **onutil** CHECK INDEX command. This command checks the order of key values and the consistency of horizontal and vertical node links for B-tree indexes and bitmaps that are associated with the specified table and provides information about how efficiently the index is using its allocated space.

By default, the **onutil** CHECK INDEX command opens all index fragments with the locking granularity defined for the table. However, if you add the LOCK keyword to the **onutil** CHECK INDEX command, it uses *intent shared locks* instead. An intent shared lock allows other users to insert or modify data in the indexed table while the index is being checked, but it does not permit dropping or altering the table during the index check.

Because users can insert and modify data while the index is being checked with an intent shared lock, the index might contain some inconsistencies even after it is checked. To make sure that an index is completely consistent, do not use the LOCK keyword.

For example, to use intent shared locks while you check the index **custidx** in the database **YR97**, enter the following command:

```
onutil
1> check index database yr97 index custidx LOCK;
```

For detailed information about the **onutil** CHECK INDEX command, refer to the *Administrator's Reference*.

Improving Performance for Index Builds

Whenever possible, the database server uses parallel processing to improve the speed of index builds. The number of parallel processes depends primarily on the number of fragments in the index. For more information, see "Parallel Index Builds" on page 11-19.

You can often improve the performance of an index build by taking the following steps:

1. Make sure that PDQPRIORITY is set to an appropriate value, as described in "PDQPRIORITY" on page 4-22.
2. Make sure that enough memory and temporary space are available to build the entire index:
   a. Estimate the amount of virtual shared memory that the database server might need for sorting.

      The following section, "Estimating Sort Memory," provides detailed information.

   b. Specify enough total DSS memory with the DS_TOTAL_MEMORY configuration parameter.

c. If not enough memory is available, estimate the amount of temporary space needed for an entire index build.

   For more information, refer to "Estimating Temporary Space for Index Builds" on page 7-24.

d. Use the **onutil** CREATE TEMP DBSPACE and CREATE TEMP DBSLICE commands to create large temporary dbspaces and specify them in the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

   For information on optimizing temporary dbspaces, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

Estimating Sort Memory

To calculate the amount of virtual shared memory that the database server might need for sorting, estimate the maximum number of sorts that might occur concurrently and multiply that number by the average number of rows in each dbspace and the average row size.

For example, if you estimate that 30 sorts could occur concurrently, the average row size is 200 bytes, and the average number of rows in a table or dbspace is 400, you can estimate the amount of shared memory that the database server needs for sorting as follows:

```
30 sorts * 200 bytes * 400 rows = 2,400,000 bytes
```

If PDQPRIORITY is `0`, the maximum amount of shared memory that the database server allocates for a sort is about 128 kilobytes for each sort thread. If PDQPRIORITY is greater than `0`, the database server allocates sort memory from the total memory allocated to the query.

Specify more memory with the DS_TOTAL_MEMORY configuration parameter and request a larger portion of that memory with the PDQPRIORITY configuration parameter. For more information, refer to "Increasing Sort Memory" on page 11-22 and "How the RGM Grants Memory" on page 12-5.

Estimating Temporary Space for Index Builds

To estimate the amount of temporary space needed for an entire index build, perform the following steps:

1. Add up the total widths of the indexed columns or returned values from user-defined functions. This value is referred to as *keysize*.
2. Estimate the size of a typical item to sort with one of the following formulas, depending on whether the index is attached or not:
   - For a nonfragmented table or a fragmented table with an attached index, use the following formula:

     ```
     sizeof_sort_item = keysize + 5
     ```
   - For fragmented tables with the index explicitly fragmented, use the following formula:

     ```
     sizeof_sort_item = keysize + 9
     ```
3. Estimate the number of bytes needed to sort with the following formula:

   ```
   temp_bytes = 2 * (rows * sizeof_sort_item)
   ```

   This formula uses the factor 2 because everything is stored twice when intermediate sort runs use temporary space. Intermediate sort runs occur when not enough memory exists to perform the entire sort in memory.

   The value for *rows* is the total number of rows that you expect to be in the table.

Chapter

8

Locking

In This Chapter

This chapter describes how the database server uses locks and how locks affect performance.

Efficient locking is an important factor for OLTP performance because of the overhead involved in obtaining a lock. Different locking strategies might be appropriate for DSS applications. Consider the locking options discussed in this chapter, and adjust locking granularity for tables and indexes to maximize concurrency but also maintain data integrity.

This chapter discusses the following topics:

- Types of locks
- Locking during query processing
- Locking during updates, deletes, and inserts
- Monitoring and configuring locks

Locking Granularity

A *lock* is a software mechanism that prevents other processes from using a resource. This chapter discusses placing locks on data. You can place a lock on the following data components:

- An individual row
- An index key
- A page of data or index keys
- A table
- A database

The amount of data that the lock protects is called *locking granularity*. Locking granularity affects performance. When a user cannot access a row or key, the user can wait for another user to unlock the row or key. If a user locks an entire page, more users might have to wait for a row in the page.

The ability of more than one user to access a set of rows is called *concurrency*. The goal of the database administrator is to increase concurrency to improve total performance without sacrificing performance for an individual user who needs a large number of locks.

Row and Key Locking

Because row and key locking are not the default behaviors, you must specify row-level locking when you create the table, as in the following example:

```
CREATE TABLE customer(customer_num serial, lname char(20)...)
   LOCK MODE ROW;
```

The ALTER TABLE statement can change the lock mode of an existing table.

When you insert or update a row, the database server creates a row lock. In some cases, you place a row lock when you read the row with a SELECT statement. When you insert, update, or delete a key, which occurs automatically when you insert, update, or delete a row, the database server creates a lock on the key in the index.

To increase concurrency, use row and key locks for good performance when applications update a relatively small number of rows. However, the database server incurs overhead whenever it obtains a lock. For an operation that requires changing a large number of rows, obtaining one lock per row might not be cost effective. In this case, consider using page locking.

Page Locking

Page locking is the default behavior when you create a table without the LOCK MODE clause.

With page locking, the database server locks the entire page that contains the row instead of locking only the row. If you update several rows on the same page, the database server uses only one lock for the page.

When you insert or update a row, the database server creates a page lock on the data page. In some cases, primarily dependent on the isolation level of the transaction, the database server creates a page lock when you simply read the row with a SELECT statement.

When you insert, update, or delete a key, which occurs automatically when you insert, update, or delete a row, the database server creates a lock on the index page that contains the key.

Important: *A page lock on an index page might decrease concurrency more significantly than a page lock on a data page. Index pages are dense and hold a large number of keys. By locking an index page, you make a potentially large number of keys unavailable to other users until you release the lock.*

Page locks are useful for tables in which the normal process changes a lot of rows at one time. For example, an orders table that holds orders that are commonly inserted and queried individually is not a good candidate for page locking. But a table that holds old orders and is updated nightly with all of the orders placed during the day might be a good candidate. In this case, the type of isolation level that you use to access the table as well as the type and duration of the lock is important. For more information refer to"Setting the Isolation Level" on page 8-9.

Table Locking

In a data warehouse environment, queries might run faster if they acquire locks of a larger granularity. For example, if a query accesses most of the rows in a table, its efficiency increases if it acquires a smaller number of table locks instead of many page or row locks.

The database server places one or more locks on a table when a user executes the LOCK TABLE statement. The database server can place the following two types of table locks:

- **Shared lock.** No other users can write to the table.
- **Exclusive lock.** No other users can read from or write to the table.

An important distinction between these two types of table locks is in the actual number of locks placed:

- In shared mode, the database server places one shared lock on the table so that no updates can be performed. In addition, the database server adds locks for every row updated, deleted, or inserted.
- In exclusive mode, the database server places only one exclusive lock on the table, no matter how many rows it updates. If you are updating most of the rows in the table, you should place an exclusive lock on the table.

Important: *A table lock on a table can decrease update concurrency radically. Only one update transaction can access that table at any given time, and that update transaction locks out all other transactions. However, multiple read-only transactions can simultaneously access the table. This behavior is useful in a data warehouse environment where the data is loaded and then queried by many users.*

Tables can be switched back and forth between table-level locking and the other levels of locking. The ability to switch locking levels is useful when you use a table for DSS queries during certain time periods but not in others.

The user requests table-level locking for a table with one of the following SQL statements:

- LOCK TABLE statement
- LOCK MODE TABLE option with the CREATE TABLE or ALTER TABLE statement

Using the LOCK TABLE Statement

The LOCK TABLE statement is used in the application. The application specifies the type of lock (shared or exclusive) to place on the table.

The following example places an exclusive lock on the table:

```
LOCK TABLE tab1 IN EXCLUSIVE MODE
```

The following example places a shared lock on the table:

```
LOCK TABLE tab1 IN SHARE MODE
```

Using the LOCK MODE TABLE Option

When a transaction accesses a table that has table-level locking, it automatically acquires a table lock if the isolation level for the transaction requires any locks at all. The user does not need to acquire a table lock explicitly with the LOCK TABLE statement.

To create a table with table-level locking use the CREATE TABLE statement, as in the following example:

```
CREATE TABLE tab1 (col1 ...) LOCK MODE TABLE
```

After you create a table, you can change its locking level with the ALTER TABLE statement, as in the following example:

```
ALTER TABLE tab2 LOCK MODE TABLE
```

When you use this LOCK MODE TABLE option, the database server automatically acquires the correct type of lock (shared or exclusive), depending on the isolation level of the transaction:

- Exclusive table locks (for write or read-write access) at all isolation levels
- Shared table locks (for read access) at Cursor Stability and Repeatable Read isolation levels

When the Database Server Locks the Table

In some cases, the database server places its own table locks. For example, if the isolation level is Repeatable Read, and the database server has to read a large part of the table, it places a table lock automatically instead of setting row or page locks. The database server also places a table lock on a table when it creates or drops an index.

Database Locking

You can place a lock on the entire database when you open the database with the DATABASE statement. A database lock prevents read or update access by anyone but the current user.

The following statement opens and locks the **sales** database:

```
DATABASE sales EXCLUSIVE
```

Setting COARSE Locking for Indexes

When an index is locked in COARSE mode, the database server places a shared lock on the index partition instead of using the key- or page-level locking specified when the index was created. Locking the entire partition reduces lock overhead, which is a critical factor in OLTP applications. Shared locks allow many read-only transactions to access the index partition at the same time.

If applications rarely update the key columns in a table, increase locking efficiency by setting the lock mode for its indexes to COARSE.

Important: *If a user updates the index and associated table when the index is locked in COARSE mode, the updating user sets an exclusive lock on the table through the index. Other users cannot access the table until the updating user releases the lock.*

When the lock mode for an index is set to COARSE, the database server first acquires an exclusive lock on the table to allow current transactions that are executing with a finer-grain lock on the table to complete their work. Then it switches the index lock mode to COARSE.

The syntax for changing the lock mode on an index is as follows:

```
ALTER INDEX index_name LOCK MODE COARSE
```

To reset the index to its original lock mode, use the ALTER INDEX statement and set the lock mode to NORMAL.

You can easily switch the lock mode between COARSE and NORMAL to accommodate periods when changes are made to key columns in the table. The index lock mode is displayed in the **onutil** CHECK INDEX output.

Waiting for Locks

When a user process encounters a lock, the default behavior of the database server is to return an error to the application immediately.

You can execute the following statement to wait indefinitely for a lock:

```
SET LOCK MODE TO WAIT
```

You can also wait for a specific number of seconds, as the following example shows:

```
SET LOCK MODE TO WAIT 20
```

To return to the default behavior (no waiting for locks), execute the following statement:

```
SET LOCK MODE TO NOT WAIT
```

Consider the performance implications of allowing sessions to wait indefinitely for locks to be released. You can prevent deadlocks if sessions wait only a specified number of seconds for a lock to be released. It is often more efficient to reenter an OLTP transaction than to slow input with long waits.

Locking with the SELECT Statement

The type and duration of locks that the database server places depend on what isolation level is set in the application and whether the SELECT statement is in an update cursor. The following section explains how isolation levels and update cursors affect locking behavior.

Setting the Isolation Level

The number and duration of locks placed on data during a SELECT statement depend on the isolation level that the user sets. The isolation level can affect overall performance because it affects concurrency.

You can set the isolation level with the SET ISOLATION or the ANSI SET TRANSACTION statement and a specific isolation-level name before you execute the SELECT statement. You can execute SET TRANSACTION only once in a transaction. You can execute SET ISOLATION more than once in a transaction, and it permits an additional isolation level, Cursor Stability. For more information, see "Cursor Stability Isolation" on page 8-11.

Dirty Read Isolation

Dirty Read isolation (or ANSI Read Uncommitted) places no locks on any rows fetched during a SELECT statement. Dirty Read isolation is appropriate for STATIC tables that are used for queries.

Use Dirty Read with care if update activity occurs at the same time as queries. With Dirty Read, the user can read a row that has not been committed to the database and might be removed or changed during a rollback. For example, consider the following scenario:

```
User 1 starts a transacion.
User 1 inserts row A.
User 2 reads row A.
User 1 rolls back row A.
```

In this case, user 2 reads a row that user 1 rolls back seconds later. In effect, user 2 has read a row that was never committed to the database. Sometimes known as a *phantom row*, uncommitted data that is rolled back can pose a problem for applications.

Because the database server does not check or place any locks for queries, Dirty Read isolation offers the best performance of all isolation levels. However, because of potential problems with phantom rows, use it with care.

Because the phantom-row problem is associated only with transactions, non-logging tables, which do not allow transactions, use Dirty Read as a default isolation level.

Committed Read Isolation

Committed Read isolation (or ANSI Read Committed) removes the problem of phantom reads. A reading process with this isolation level checks for locks before it returns a row. Because inserted or updated rows are locked until the transaction commits, the reading process does not return any uncommitted rows.

The database server does not place any locks for rows read during Committed Read. It only checks the lock table for any existing locked rows.

Committed Read is the default isolation level for databases with logging, and it is an appropriate isolation level for most activities.

Cursor Stability Isolation

A reading process with Cursor Stability isolation acquires a shared lock on the row that is currently fetched. This action ensures that no other process can update the row until the reading process fetches a new row.

The ISOLATION_LOCKS configuration parameter determines the number of rows to lock when Cursor Stability isolation level is in effect. ISOLATION_LOCKS offers an intermediate solution between Repeatable Read, which locks the entire table, and Committed Read, which does not obtain read locks. Increasing ISOLATION_LOCKS allows more efficient row buffering, but locking many rows can reduce concurrency for the table. For information about the ISOLATION_LOCKS parameter, refer to"ISOLATION_LOCKS" on page 5-18.

The pseudocode in Figure 8-1 shows when the database server places and releases locks with a cursor.

If you do not use a cursor to fetch data, Cursor Stability isolation behaves in the same way as Committed Read. No locks are actually placed.

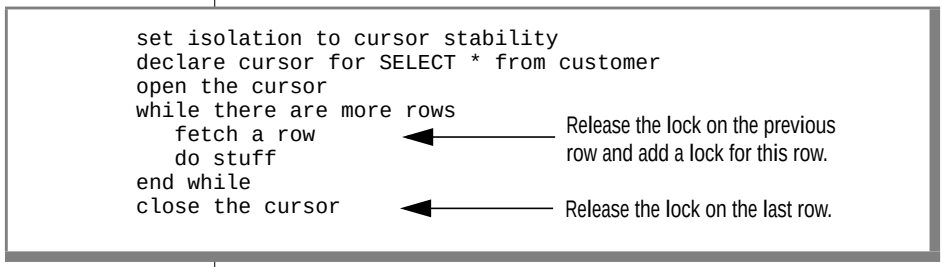


Figure 8-1
Locks Placed for Cursor Stability

Repeatable Read Isolation

Repeatable Read isolation, which is the equivalent of ANSI Serializable and ANSI Repeatable Read, is the most strict isolation level. With Repeatable Read, the database server locks all rows examined, not just rows fetched, for the duration of the transaction.

The pseudocode in Figure 8-2 shows when the database server places and releases locks with a cursor.

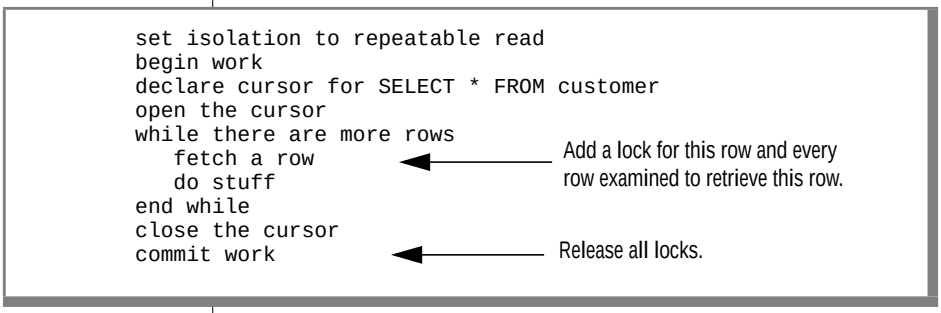


Figure 8-2
Locks Placed for Repeatable Read

Repeatable Read is useful during any processing in which multiple rows are examined and none must change during the transaction. For example, consider an application that checks the account balance of three accounts that belong to one person. The application gets the balance of the first account and then the second. But, at the same time, another application begins a transaction that debits the third account and the credits the first account. By the time that the original application obtains the account balance of the third account, it has been debited. However, the original application did not record the debit of the first account.

When you use Committed Read or Cursor Stability, the previous situation can occur. However, with Repeatable Read, it cannot. The original application holds a read lock on each account that it examines until the end of the transaction, so the application trying to change the first and third account would fail (or wait, depending upon SET LOCK MODE).

Because even examined rows are locked, if the database server reads the table sequentially, many rows unrelated to the query result can be locked. For this reason, use Repeatable Read isolation for tables when the database server can use an index to access a table.

Locking and Update Cursors

An update cursor is a special kind of cursor that applications can use when a selected row might be updated. To use an update cursor, execute SELECT FOR UPDATE in your application.

Update cursors use *promotable locks*. With a promotable lock, when the application fetches the row, the database server places an update lock so that other users can still view the row, but the lock is changed to an exclusive lock when the application uses an update cursor and the UPDATE...WHERE CURRENT OF statement to update the row.

The advantage of using an update cursor is that other users cannot view or change the row that you are viewing while you are viewing it and before you update it.

If you do not update the row, the default behavior of the database server is to release the update lock when you execute the next FETCH statement or close the cursor. However, if you execute the SET ISOLATION statement with the RETAIN UPDATE LOCKS clause, the database server does not release any currently existing or subsequently placed update locks until the end of the transaction.

The pseudocode in Figure 8-3 shows when the database server places and releases update locks with a cursor. The database server releases the update lock on row one as soon as the next fetch occurs. However, after the database server executes the SET ISOLATION statement with the RETAIN UPDATE LOCKS clause, it does not release any update locks until the end of the transaction.

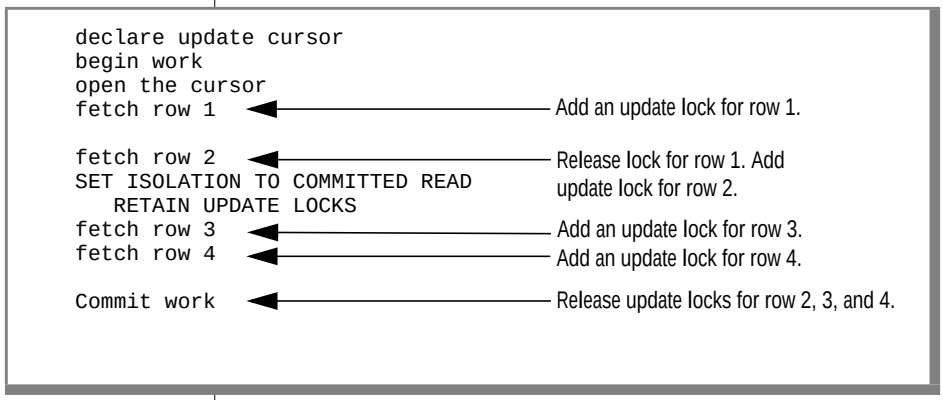


Figure 8-3
When Update Locks Are Released

In an ANSI-compliant database, you usually do not need update cursors because any select cursor behaves the same as an update cursor without requiring the RETAIN UPDATE LOCKS clause.

The pseudocode in Figure 8-4 shows when the database server places and releases locks with a cursor.

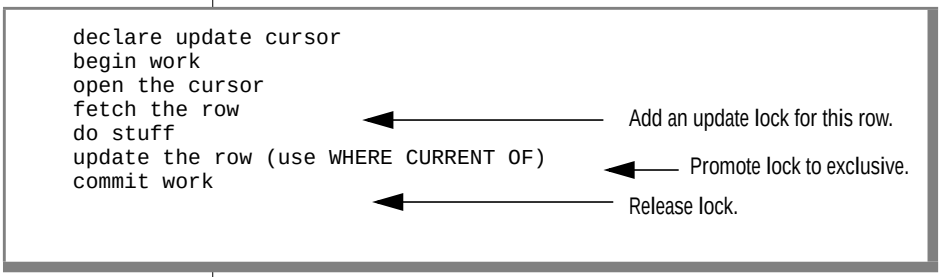


Figure 8-4
When Update Locks Are Promoted

For detailed information about the RETAIN UPDATE LOCKS clause of the SET ISOLATION LEVEL statement, see the *Informix Guide to SQL: Syntax*.

Placing Locks with INSERT, UPDATE, and DELETE

When you execute an INSERT, UPDATE, or DELETE statement, the database server uses exclusive locks. With an exclusive lock, no other users can view a row unless they are using the Dirty Read isolation level. In addition, no other users can update or delete the item until the database server removes the lock.

Key-Value Locking

When a user deletes a row in a transaction, the row cannot be locked because it does not exist. However, the database server must somehow record that a row existed until the end of the transaction.

The database server uses *key-value locking* to lock the deleted row. When the database server deletes a row, it does not remove key values in the indexes for the table immediately. Instead, it marks each key value as deleted and places a lock on the key value.

If other users encounter key values that are marked as deleted, the database server determines whether a lock exists. If a lock exists, the delete has not been committed. The database server sends a lock error back to the application, or it waits for the lock to be released if the user executed SET LOCK MODE TO WAIT.

One of the most important uses for key-value locking is to assure that a unique key stays unique until the end of the transaction in which it is deleted. Without this protection mechanism, user A might delete a unique key in a transaction. Before the transaction commits, user B might insert a row with the same key, which would make rollback by user A impossible. Key-value locking prevents user B from inserting the row until the end of user A's transaction.

Monitoring and Administering Locks

The database server stores lock information in an internal lock table. When the database server reads a row, it checks to see if the row or its associated page, table, or database is listed in the lock table. If it is in the lock table, the database server also checks the lock type. The lock table can contain the following types of locks.

Lock Name	Description	Statement That Usually Places the Lock
S	Shared lock	SELECT
X	Exclusive lock	INSERT, UPDATE, DELETE
U	Update lock	SELECT in an update cursor
B	Byte lock	Any statement that updates VARCHAR columns

In addition, the lock table might store *intent locks* with the same lock type. A a process uses an intent lock to register its possible intent to lock an item, so that other processes cannot place a lock on the item but can read it.

Depending on the type of operation and the isolation level, the database server might continue to read the row and place its own lock on the row, or it waits for the lock to be released if the user executed SET LOCK MODE TO WAIT. The following table shows what locks a process can place if another process holds a certain type of lock. For example, if one process holds an exclusive lock on an item, another process that requests any kind of lock (exclusive, update or shared) receives an error.

	Hold X Lock	Hold U Lock	Hold S Lock
Request X lock	No	No	Yes
Request U lock	No	No	Yes
Request S lock	No	Yes	Yes

Monitoring Locks

To view the lock table, use **onstat -k**, or **xctl onstat -k** for multiple coservers. Figure 8-5 shows sample output for **onstat -k**.

```
Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
300b77d0 0        40074140 0        HDR+S    10002    106        0
300b7828 0        40074140 300b77d0 HDR+S    10197    123        0
300b7854 0        40074140 300b7828 HDR+IX   101e4    0          0
300b78d8 0        40074140 300b7854 HDR+X    101e4    102        0
 4 active, 5000 total, 8192 hash buckets
```

Figure 8-5
onstat -k Output

In this example, a user is inserting one row in a table. The user holds the following locks, described in the order shown:

- A shared lock on the database
- A shared lock on a row in the **systables** system catalog table
- An intent-exclusive lock on the table
- An exclusive lock on the row

To find out which table has locks, execute the following SQL statement. Substitute the value shown in the **tblsnum** field in the **onstat -k** output for *tblsnum*.

```
SELECT tabname
FROM systables
WHERE partnum = hex(tblsnum)
```

For a complete description of **onstat -k** output, refer to the *Administrator's Reference*.

Configuring and Monitoring the Number of Locks

The LOCKS configuration parameter controls the size of the internal lock table. If the number of locks placed exceeds the value set by LOCKS, the application receives an error. For more information on how to determine an initial value for the LOCKS configuration parameter, refer to "LOCKS" on page 4-19.

To specify the number of rows that can be locked in Cursor Stability isolation level, set the ISOLATION_LOCKS configuration parameter to a value greater than 1. Make sure that the LOCKS configuration parameter is set correctly if you set ISOLATION_LOCKS. For more information, refer to "Cursor Stability Isolation" on page 8-11.

To monitor the number of times that applications receive the out-of-locks error, view the **ovlock** field in the output of **onstat -p** (or **xctl onstat -p** for multiple coservers).

If the database server often receives the out-of-locks error, you might increase the LOCKS parameter value. However, a very large lock table can slow performance. Although the algorithm for reading the lock table is efficient, you incur some cost for reading a large table each time that the database server reads a row. If the database server is using an unusually large number of locks, examine how individual applications are using locks.

First, monitor sessions with **onstat -u** to see if a particular process is using an especially high number of locks (the high value in the **locks** column). If one session uses a large number of locks, examine the SQL statements in the application to determine whether you should lock the table or use individual row or page locks.

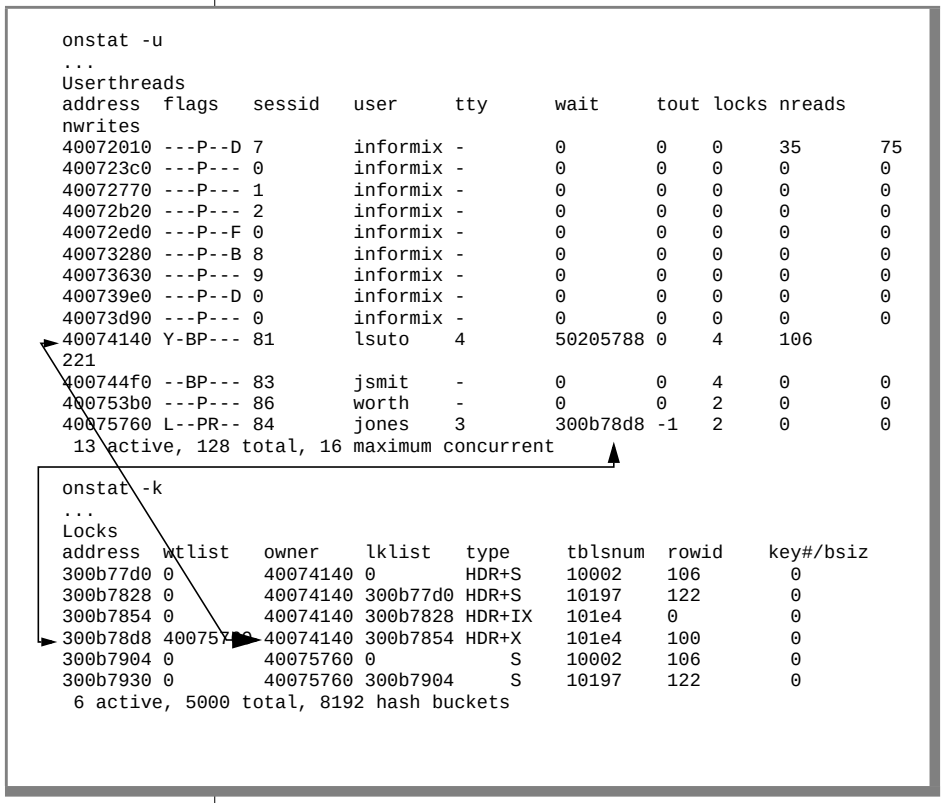
A table lock is more efficient than individual row locks, but it reduces concurrency. To reduce the number of locks placed on a table, you can also alter a table to use page locks instead of row locks. However, page locks reduce overall concurrency for the table, which can affect performance.

Monitoring Lock Waits and Lock Errors

If the application executes SET LOCK MODE TO WAIT, the database server waits for a lock to be released instead of returning an error. An unusually long wait for a lock can make users think that the application is hanging.

In Figure 8-5, the **onstat -u** output shows that session ID 84 is waiting for a lock (L in the first column of the **Flags** field). The address field shows the address of the lock for which the user is waiting. To find out the owner of the lock, use the **onstat -k** command.You can cross-reference the owner of the lock back to the **onstat -u** output. In the example, session ID 81 is the owner of the lock.

To eliminate the contention problem, you can have the user of session 81 exit from the application. If this action is not possible, you can kill the application process or remove the session with **onmode -z**.



```
onstat -u
...
Userthreads
address  flags   sessid   user     tty      wait     tout locks nreads
nwrites
40072010 ---P--D 7        informix -        0        0    0     35       75
400723c0 ---P--- 0        informix -        0        0    0     0        0
40072770 ---P--- 1        informix -        0        0    0     0        0
40072b20 ---P--- 2        informix -        0        0    0     0        0
40072ed0 ---P--F 0        informix -        0        0    0     0        0
40073280 ---P--B 8        informix -        0        0    0     0        0
40073630 ---P--- 9        informix -        0        0    0     0        0
400739e0 ---P--D 0        informix -        0        0    0     0        0
40073d90 ---P--- 0        informix -        0        0    0     0        0
40074140 Y-BP--- 81       lsuto    4        50205788 0    4     106
 221
400744f0 --BP--- 83       jsmit    -        0        0    4     0        0
400753b0 ---P--- 86       worth    -        0        0    2     0        0
40075760 L--PR-- 84       jones    3        300b78d8 -1   2     0        0
 13 active, 128 total, 16 maximum concurrent

onstat -k
...
Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
300b77d0 0        40074140 0        HDR+S    10002    106        0
300b7828 0        40074140 300b77d0 HDR+S    10197    122        0
300b7854 0        40074140 300b7828 HDR+IX   101e4    0          0
300b78d8 40075760 40074140 300b7854 HDR+X    101e4    100        0
300b7904 0        40075760 0            S    10002    106        0
300b7930 0        40075760 300b7904     S    10197    122        0
 6 active, 5000 total, 8192 hash buckets
```

Figure 8-6
onstat -u Output That Shows Lock Use

Monitoring Deadlocks

A *deadlock* occurs when user processes hold locks that other users want to acquire.

For example, user **joe** holds a lock on row 10. User **jane** holds a lock on row 20. Suppose that **jane** wants to place a lock on row 10, and **joe** wants to place a lock on row 20. If both users execute SET LOCK MODE TO WAIT, they might wait for each other forever.

If user processes access tables or fragments on the local coserver, the database server uses the lock table to detect deadlocks automatically and stop them before they occur. Before a lock is granted, the database server examines the lock list for each user. If a user holds a lock on the resource that the requestor wants to lock, the database server traverses the lock wait list for the user to see if the user is waiting on any locks that the requestor holds. If so, the requestor receives an deadlock error.

Deadlock errors in OLTP applications can be unavoidable if applications update the same rows frequently. However, certain applications might always be in contention with each other. Examine applications that are producing a large number of deadlocks and try to run them at different times. To monitor the number of deadlocks, use the **deadlks** field in the output of **onstat -p** (or **xctl onstat -p** for multiple coservers).

To monitor the number of distributed deadlock timeouts, use the **dltouts** field in the **onstat -p** output.

Reducing Deadlocks

Deadlocks often occur in OLTP systems in which the same table or table fragment is updated or read with locking by many users almost simultaneously. For this reason, you should resolve deadlocks automatically and immediately so that they do not slow or halt the system.

In DSS databases, deadlocks might also occur if many queries are reading the same table. In DSS databases, however, many tables are assumed to be in a stable state, so they can be STATIC tables, which are not locked when they are read.

To reduce the number of deadlocks in OLTP databases, you might use the following methods:

- Review table fragmentation.

  Use the deadlock information in the log to determine which tables and fragments are associated with most deadlocks and consider a different fragmentation strategy. You might make the fragmentation granularity smaller.

- Review the isolation level of transactions.

  For information about the relation of the isolation level and locking, see "Setting the Isolation Level" on page 8-9.

  Make sure that SPL routines used in transactions specify the appropriate isolation level.

- Review the table and locking mode.

  Unless transactions change several rows at a time, set STANDARD tables to row-level locking for OLTP applications.

  Make as many tables as possible STATIC tables, which do not permit changes but also do not require locks of any kind. Tables that contain information that does not change often or changes only at regular intervals, such as product tables, department tables, and so on, can be made STATIC tables. You can change the table mode when updates are required, and updates might be applied in a batch process.

Chapter

9

Fragmentation Guidelines

In This Chapter

This chapter discusses how table and index fragmentation can reduce data contention and allow fragments to be eliminated for efficient query processing.

Fragmenting tables and indexes across coservers and disks can significantly reduce I/O contention for data and improve parallel processing. Additional performance improvement results if you construct the fragmentation scheme in such a way that unnecessary fragments can be eliminated in query processing.

This chapter discusses the following topics:

- Planning a fragmentation strategy
- Fragmenting with cogroups and dbslices
- Designing a distribution scheme
- Eliminating table fragments in query processing
- Fragmenting indexes
- Fragmenting temporary tables
- Improving the performance of attaching and detaching fragments
- Monitoring fragmentation

For information about how fragmentation affects parallel query execution, refer to Chapter 11, "Parallel Database Query Guidelines."

This manual emphasizes the fragmentation methods that are most useful for performance improvements. For an introduction to general fragmentation methods, refer to the *Informix Guide to Database Design and Implementation*.

For information about the SQL statements that create fragmented tables, refer to the *Informix Guide to SQL: Syntax*.

Planning a Fragmentation Strategy

Planning a fragmentation strategy requires you to make the following decisions:

1. Identify your primary fragmentation goal.

   Your fragmentation goals depend on the types of applications and the design of queries and transactions that access the table.

2. Analyze the workload.

   OLTP applications and DSS queries might require different fragmentation strategies. For example, OLTP applications might require tables fragmented so that many users can access them simultaneously. DSS queries might require tables fragmented for improved efficiency of parallel processing.

   Consider questions such as the following ones:

   - What tables do queries and transactions access most often?
   - What attributes are used in SELECT or WHERE clauses?
   - What tables do queries join most often?
   - For updates, what fields are changed most often?
   - How often are rows added to or deleted from tables?

3. Decide how the table should be fragmented.

   You must make the following decisions:

   - Whether to fragment the table data, the table index, or both
   - What the most useful distribution of rows or index keys is for the table

4. Choose one of the five distribution schemes.

   If you choose an expression-based, range, or hybrid distribution scheme, you are responsible for designing suitable fragment expressions.

   If you choose a system-defined hash or round-robin distribution scheme, the database server determines which rows to put in a specific fragment.

5. To complete the fragmentation strategy, decide on the location of the fragments.

   The number of coservers, dbslices, and dbspaces across which you fragment the table determines the number of fragments.

Identifying Fragmentation Goals

Analyze your application and workload to determine how to balance the following fragmentation goals:

- Improved performance for queries and transactions

  To improve the performance for OLTP transactions, fragment tables so that the requested rows can be accessed immediately from the appropriate table fragment and many users can access different fragments of the table simultaneously.

  For DSS queries, data required by the query can be scanned in parallel by one thread for each table fragment to improve query performance. Certain fragmentation schemes allow the database server to identify fragments that are not required by the query and to skip them automatically.

- Reduced disk contention and disk bottlenecks

  When tables are fragmented across several disks and coservers, many users and many queries can access separate fragments without causing I/O bottlenecks or data contention.

- Increased data availability

  Fragmentation can improve data availability if devices fail. Table fragments on the failed device can be restored quickly, and other fragments are still accessible.

- Increased granularity for backup and restore

  Consider how restoring tables and fragments of tables might affect transaction and query processing and whether warm or cold restores would be required to restore backed-up data.

- Improved data-load performance

  When the database server uses parallel inserts and external tables to load a table in express mode, if the table is fragmented across multiple coservers, the database server allocates threads to write data into the fragments in parallel using light append. For more information about loading data from external tables, refer to the *Administrator's Reference*.

  You can also use the ALTER FRAGMENT ON TABLE statement with the ATTACH clause to add data quickly to a very large table. For more information, refer to "Attaching and Detaching Table Fragments" on page 9-62.

The following factors primarily govern the performance of a fragmented table:

- How disk space is allocated in dbslices for fragments, discussed in "Creating Cogroups and Dbslices for Fragmentation" on page 9-17
- The distribution scheme used to assign rows to individual fragments, discussed in "Designing a Distribution Scheme" on page 9-23

Improving Query Performance

If the primary goal of fragmentation is improved performance for DSS queries, distribute the rows of the table evenly over the different coservers. Overall query-completion time is reduced when the database server does not have to wait during parallel processing of queries while one thread retrieves data from a table fragment that has more rows than other fragments. If you use the fact-dimension database model, your fact table should be fragmented across coservers. Consider using hybrid fragmentation, described on "Creating a Hybrid Distribution Scheme" on page 9-34, to fine-tune fragmentation for major tables.

One exception to fragmenting tables evenly across coservers is a dimension table that contains fewer than 1000 rows. Such tables are usually more efficient when they are stored on a single coserver.

If you use round-robin fragmentation, do not fragment the index. Consider placing that index in a separate dbspace from other table fragments.

For more information about improving performance for queries, see "Designing Distribution for Fragment Elimination" on page 9-41 and Chapter 10, "Queries and the Query Optimizer."

Reducing I/O Contention

Fragmentation can reduce contention for data in tables. Fragmentation often reduces contention when many simultaneous queries against a table perform index scans to return a few rows, such as queries about the inventory of specific items in an order-entry application.

For tables that are subjected to this type of query workload, fragment both the index keys and data rows with one of the following distribution schemes to allow each query to eliminate unneeded fragments from its scan:

- Expression-based
- System-defined hash
- Hybrid
- Range

For more information, refer to "Designing Distribution for Fragment Elimination" on page 9-41.

To fragment a table for reduced contention, start by investigating which queries and transactions access which rows of the table. Next, fragment data so that some of the queries and transactions are routed to one fragment while others access a different fragment. The database server performs this routing when it evaluates the fragmentation rule for the table. Finally, store the fragments on separate coservers and disks.

Your success in reducing contention depends on how much you know about the distribution of data in the table and how queries access the data.

For example, if queries against the table access rows at roughly the same rate or randomly, try to distribute rows evenly across the fragments. However, if certain rows are accessed more often than others, try to distribute these rows over the fragments to balance the access. For more information, refer to "Creating an Expression-Based Distribution Scheme" on page 9-32.

Increasing Data Availability

When you distribute table and index fragments across different disks, you improve the availability of data during disk failures. If you have turned DATASKIP on, the database server can continue to allow access to fragments stored on disks that remain operational.

To specify which fragments can be skipped, execute the SET DATASKIP statement before you execute a query. You can also set the DATASKIP configuration parameter. Use the **onstat -f** utility to find out if DATASKIP is on or off. If DATASKIP is on, the **onstat -f** output lists the dbspaces that can be skipped if they are not available. This feature has important implications for the following types of applications:

- Applications that do not require access to all fragments

  An OLTP application or a query that does not require the database server to access data in an unavailable fragment can still successfully retrieve data from fragments that are available. For example, if the table is fragmented by expression on a single column, as described in "Creating an Expression-Based Distribution Scheme" on page 9-32, the database server can determine if a row is contained in a fragment without accessing the fragment. If the query accesses only rows that are contained in available fragments, a query can succeed even when some of the data in the table is unavailable.

- Applications that accept the unavailability of data

  Some applications might be designed in such a way that they can accept the unavailability of data in a fragment and retrieve only the data that is available. For example, some decision-support queries require only a statistical sample of the table data.

If your fragmentation goal is increased availability of data, fragment both table rows and index keys so that if a disk drive fails, some of the data is still available.

If applications must always be able to access a particular subset of your data, keep those rows together in the same mirrored dbspace.

Increasing Granularity for Backup and Restore

Consider the following two backup and restore factors when you decide how to distribute dbspaces across disks and coservers:

- **Data availability.** When you decide where to place your tables or fragments, remember that if a device that contains a dbspace fails, all tables or table fragments in that dbspace are inaccessible even though tables and fragments in other dbspaces are accessible. The need to limit data unavailability in the event of a disk failure might influence which tables you group together.
- **Cold versus warm restores.** Although you must perform a cold restore if a dbspace that contains critical data fails, you need to perform only a warm restore if a noncritical dbspace fails.

  Reduce the impact of cold restores by careful choice of the dbspace where you store critical data, which includes the root dbspace on each coserver and all dbspaces that contain logical or physical logs. If possible, store critical data on separate disks.

For more information about backup and restore, see the *Backup and Restore Guide*.

Evaluating Fragmentation Factors for Performance

As you formulate your fragmentation strategy, keep these interrelated factors in mind:

- Even balance of processing across all coservers
- Parallel-processing advantages for tables that are fragmented across coservers
- Increased query-processing speed if the database server can eliminate fragments

Balancing Processing Across All Coservers

For decision-support applications, the primary fragmentation consideration should be to balance processing evenly across all coservers. When one or two coservers are working harder than others, the entire system slows down.

To take advantage of the hardware parallelism that underlies the database server, SQL operations execute in parallel. If the workload is not evenly distributed across coservers, parallel execution is unbalanced, and performance suffers.

All of the performance-related factors discussed in this manual affect processing balance, some more than others. Although the importance of some of these factors depends on the kind of workload on the database server, you should analyze queries and transactions and fragment tables to avoid *data skew* as much as possible. Data skew occurs when a disproportionate amount of the data that an SQL operation requires resides on one coserver instead of being distributed evenly across all coservers. Consider expression and hybrid fragmentation schemes to distribute data appropriately for the queries that users run most often.

Balance the resources on all nodes where participating coservers reside. All participating coservers should have the same number of processors and disks per processor and the same amount of memory.

Nodes that are used as connection coservers for client applications should have additional ports, however, and might use processor affinity and allow additional memory to ensure processor balancing for the database server workload. If incoming data or data preprocessing is intensive, you might use a dedicated node that does not run a database server for that purpose.

Fragmenting Tables Across Coservers

Because coservers process data in parallel, parallel I/O performance improves in a linear fashion as table fragments are added if the fragments are added evenly across coservers. This performance improvement is limited by the number of CPUs, the latency of the high-speed interconnect between coservers, and the bus bandwidth.

Increasing the number of table fragments can reduce contention for some OLTP applications. For DSS applications, however, as the number of rows in a fragment decreases, a query search is increasingly likely to span multiple fragments. If the number of rows in each fragment is small, the overhead for coordinating scan operations across coservers cancels the gains from fragmentation unless tables are fragmented to take advantage of collocated joins and can eliminate many fragments. For more information, refer to "Ensuring Collocated Joins" on page 9-30.

Eliminating Fragments for Fast Queries and Transactions

For DSS queries, you can reduce the number of I/O operations if queries can take advantage of fragment elimination.

Before you decide on a fragmentation strategy, identify the tables that large queries require and determine what portions of each table these queries actually examine. Then fragment the tables to restrict the scope of queries to a subset of table fragments or distribute fragments by hash on the column that the queries use most often.

Fragment elimination is even more important for efficient OLTP transaction processing. Examine transactions and fragment tables so that many transactions against the same table can occur on different fragments of the table.

When you eliminate fragments from a scan, you eliminate the associated I/O operations and delays, and you also reduce demand for buffers and LRU queue activity. To perform calculations associated with the query or to support other queries or OLTP operations, the database server can use CPU cycles that are otherwise used to scan fragments and manage buffers. For detailed information about creating queries that eliminate fragments, see "Designing Distribution for Fragment Elimination" on page 9-41.

Base your decision regarding the number of fragments in a table on how you distribute your data, or distribute your data based on the number of coservers and disks that are available. The most important factor in planning a fragmentation strategy is careful analysis of typical queries and transactions.

Examining Your Data and Queries

To determine a fragmentation strategy, you must know how the data in a table is used.

To gather information about the relation between queries and tables

1. Identify the queries that are critical to performance and whether they are OLTP or DSS.
2. Use the SET EXPLAIN statement to find out how the data is being accessed. For more information on how to interpret the output of the SET EXPLAIN statement, refer to Chapter 10, "Queries and the Query Optimizer." Sometimes you can find out how the data is accessed by reviewing the SELECT statements together with the table schema.
3. Find out if queries access data randomly or if a pattern exists. To reduce I/O contention, use information about the access pattern to fragment tables. You might be able to use this information to improve ad hoc queries generated by a third-party application that you cannot manage.
4. Find out if certain tables are always joined in DSS queries. Fragment joined tables across the same coservers to take better advantage of parallel processing.
5. Find out what statements create temporary files. Because decision-support queries often create and access large temporary files, placement of temporary dbspaces can be critical to performance.
6. Examine the columns in the table to decide which fragmentation scheme would keep all scan threads equally busy for the decision-support queries. To see how the column values are distributed, execute the UPDATE STATISTICS statement and examine the data distribution with **dbschema**:

   ```
   dbschema -d database -hd table
   ```

Planning Storage Spaces for Fragmented Tables and Indexes

When you fragment a table or index, the physical placement issues that pertain to an unfragmented table or applies to individual fragments. For more details on table placement issues, refer to Chapter 6, "Table Performance."

Tip: *If each node contains more than one coserver, the dbspaces for each coserver should be on separate disks. Each disk should be assigned to a single coserver to prevent coservers from sharing disks. If the nodes use disk clusters or disk arrays, however, you might not be able to restrict certain disks to use by certain coservers. In that case, create chunks for dbspaces evenly across as many disks as possible within the disk cluster. The system administrator should be able to help you partition disks and create chunks in such a way that you prevent coservers from sharing disks.*

Fragmented and nonfragmented tables and indexes differ in the following ways:

- For fragmented tables, each fragment is placed in a separate, designated dbspace. For nonfragmented tables, the table can be placed in the default dbspace of the current database. Regardless of whether the table is fragmented, Informix recommends that each chunk the dbspace contains be created on a separate disk if possible.
- Extent sizes for a fragmented table are usually smaller than the extent sizes for an equivalent nonfragmented table because fragments do not grow in increments as large as the entire table. For more information on extent sizes for fragmented tables, refer to "Choosing Extent Sizes" on page 6-22.
- In a fragmented table, the row identifier is not an unchanging pointer to the row on a disk. A row identifier for a fragmented table is made up of a fragment ID and row identifier. Although these two fields are unique, they can change during the life of the row. An application cannot access the row by fragment ID and row identifier. The database server keeps track of the fragment ID and row identifier combination and uses it internally to point to a table row in a fragment.

- An attached index on a fragmented or nonfragmented table requires 4 bytes for the row identifier. A detached index requires 8 bytes of disk space per key value for the fragment ID and row identifier combination. For more information on how to estimate space for an index, refer to "Estimating Table Size" on page 6-15.

Decision-support queries usually create and access large temporary files. Placement of temporary dbspaces is a critical factor for performance. For more information about placement of temporary files, refer to "Creating and Specifying Dbspaces for Temporary Tables and Sort Files" on page 9-60.

Table and index fragments are placed in separate tblspaces with their own extents and *tblspace ID*s. The tblspace ID is a combination of the fragment ID and partition number.

Important: *Data and index pages reside in the same dbspace but not in the same tblspace in either a fragmented table or a nonfragmented table.*

The database server stores the location of each table and index fragment, and related information, in the system catalog table **sysfragments**. Use **sysfragments** to access the following information about fragmented tables and indexes:

- The value in the **fragtype** field specifies the fragment type of the object, such as **T** for a table fragment, **I** for an index fragment, or **B** for a TEXT or BYTE data fragment.
- The value in the **partn** field is the physical location identifier of the fragment.
- The value in the **strategy** field is the distribution scheme used in the fragmentation strategy.

For a complete description of field values that the **sysfragments** system catalog table contains, refer to the *Informix Guide to SQL: Reference*. For information on how to use the **sysfragments** table to monitor your fragments, refer to "Monitoring Fragmentation on a Specific Coserver" on page 9-70.

Creating Cogroups and Dbslices for Fragmentation

You can create cogroups and dbslices to:

- simplify fragmentation of very large tables across multiple coservers.
- improve performance of queries and other database operations that access tables fragmented across multiple coservers.

For information about how to fragment and manage tables, depending on your fragmentation goals, see "Designing a Distribution Scheme" on page 9-23.

The number of dbslices and dbspaces that you can create is determined by the CONFIGSIZE configuration parameter and two of its overriding configuration parameters, MAX_DBSLICES and MAX_DBSPACES. For information about these configuration parameters, refer to the *Administrator's Reference*.

Creating Cogroups and Dbslices

For ease and efficiency of management, you can group the coservers that make up the database server into *cogroups*, which are named lists of coservers, and manage each group as a logical unit. For example, you might create the cogroup **sales** to manage coservers that contain dbspaces used for your order-entry database. A predefined cogroup, **cogroup_all**, includes all coservers. For information about creating cogroups, refer to the *Administrator's Guide*.

On the coservers included in a cogroup, you create *dbslices*, which are lists of dbspaces that the database server manages as a single logical storage object. For example, you might use **onutil** to create the following customer dbslice from the **sales** cogroup:

```
% onutil
1> CREATE DBSLICE cust_dbslc
2>    FROM cogroup sales
3>    CHUNK "/dev/dbsl_customer%c"
4>    SIZE 490000;
DBslice successfully created.
```

When you create **cust_dbslc** in the previous example, the database server creates dbspaces on all **/dev/dbsl_customer** chunks that have been created across the coservers in the **sales** cogroup. As the database server creates dbspaces, it names them by combining the dbslice name and an ordinal number. In the previous example, the first dbspace created is **cust_dbslc.1**, the second is **cust_dbslc.2**, and so on. If cogroup **sales** contains coservers 5, 6, 9, and 10, the dbspaces are distributed as follows.

Dbspace	Chunk
cust_dbslc.1	/dev/dbsl_customer.5
cust_dbslc.2	/dev/dbsl_customer.6
cust_dbslc.3	/dev/dbsl_customer.9
cust_dbslc.4	/dev/dbsl_customer.10

A dbslice simplifies the creation of fragmented tables because you can refer to all of the dbspaces for a single table with a single name, the dbslice name. For example, to fragment a table across the dbspaces in the **cust_dbslc** dbslice, use the following CREATE statement to specify a single dbslice name instead of four dbspace names:

```
CREATE TABLE customer
   (cust_id integer,
   ...
   )
   FRAGMENT BY HASH (cust_id)
      IN cust_dbslc;
```

This example shows the **customer** table fragmented by system-defined hash into all of the dbspaces in the **cust_dbslc** dbslice.

For information about creating dbslices that increase the possibility of collocated joins, which join table fragments on each local coserver instead of shipping data across the interconnect, see "Creating Dbslices for Collocated Joins" on page 9-20.

Tip: *Place fragments for each important table in their own dbslice. If you place fragments of more than one table in the same dbslice, monitoring and tuning is extremely difficult.*

Increasing Parallelism by Fragmenting Tables Across Coservers

You increase the degree of parallelism when you fragment tables across multiple coservers. Fragmentation across coservers ensures that table fragments are processed in parallel by threads that are running on each coserver.

Fragmenting tables across coservers provides these advantages:

- More efficient use of shared memory

  The database server uses the resources on each coserver to process table fragments in parallel.
- More efficient fragment elimination

  Expression, range, and hash fragmentation schemes let the database server eliminate fragments for queries that use the fragmentation columns in WHERE clauses.
- Higher degree of parallelism for scans and sorts

  The virtual processors (VPs) on each coserver can process queries in parallel.
- More efficient join operations

  Collocated joins can reduce traffic between coservers.

Using Dbslices for Performance and Ease of Maintenance

Although cogroups and dbslices are primarily logical organizations that make managing a complex database server easier, they also have the following implications for performance:

- Dbslices that contain more than one dbspace on each coserver can increase data granularity and parallel processing of queries, especially if tables are fragmented with hybrid methods.
- Dbslices make it easy to distribute temporary dbspaces evenly across coservers.

- Dbslices for log files make it easy to manage logical logs across coservers.
- Dbslices permit *collocated joins*, in which rows are scanned and joined on the local coserver to reduce traffic between coservers.

Creating Dbslices for Collocated Joins

The way in which you create a dbslice to distribute the dbspaces across coservers can change the speed of queries dramatically, especially if tables are fragmented to take advantage of collocated joins. A *collocated join* is a join that is performed locally on one coserver before data is shipped to other coservers for processing.

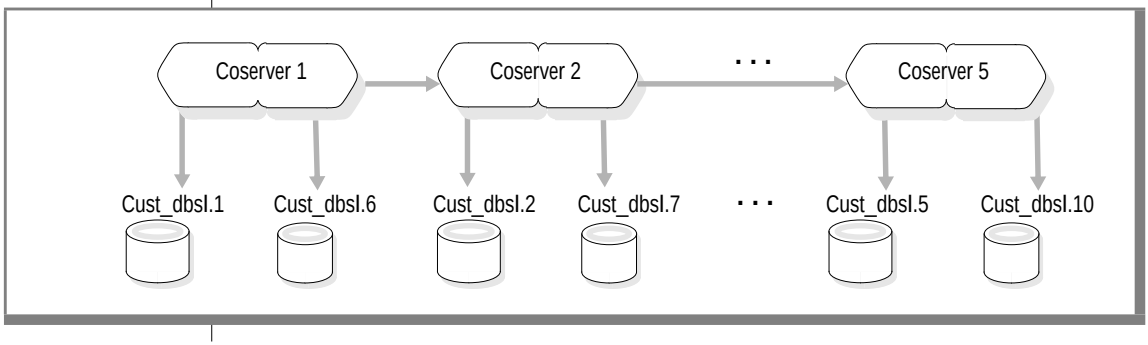
For example, you might fragment your **customer** table across five coservers with two disks on each coserver.

If you create a round-robin dbslice as in the following sample **onutil** command, you gain the advantage of collocated joins because the dbspaces are defined round-robin across coservers rather than in a coserver:

```
% onutil
1> CREATE DBSLICE cust_dbsl FROM
2> COGROUP sales CHUNK "/dbspaces/dbs1%c" SIZE 490000,
3> COGROUP sales CHUNK "/dbspaces/dbs2%c" SIZE 490000;
Dbslice successfully added.
```

This **onutil** command creates dbspaces **cust_dbsl.1** through **cust_dbsl.5** on separate coservers and dbspaces **cust_dbsl.6** through **cust_dbsl.10** on separate coservers**.** Figure 9-1 shows the dbspaces that this **onutil** CREATE DBSLICE command creates on each coserver.

Figure 9-1
Dbslice with Dbspaces Created Round-Robin Across Coservers



If you create a dbslice as in the following sample **onutil** command, queries do not have the advantage of collocated joins:

```
% onutil
1> CREATE DBSLICE cust_dbsl FROM COGROUP cust_group
2> CHUNK "/dbspaces/dbs%r(1..2)"
3> SIZE 490000;
Dbslice successfully added.
```

This dbslice creation command creates dbspaces **cust_dbsl.1** and **cust_dbsl.2** on the first coserver, dbspaces **cust_dbsl.3** and **cust_dbsl.4** on the second coserver, and so forth. Because the database server creates collocated join threads in a round-robin fashion across coservers, queries cannot take advantage of collocated joins with this dbspace layout.

For detailed information about creating cogroups and dbslices, see the *Administrator's Guide*.

Creating Dbslices to Increase Data Granularity

For very large tables, increasing the granularity of data can improve query and transaction processing dramatically.

For example, if a table uses a hybrid fragmentation scheme that fragments the table on one column by expression into a dbslice and on another column by hash across dbspaces in the dbslice, queries and transactions that select rows based on the fragmenting columns can quickly find required rows without accessing all table fragments.

The following example shows how to use the **onutil** CREATE DBSLICE command to create a dbslice across 16 coservers in the **orders_cogroup** cogroup so that each coserver dbslice section contains three dbspaces:

```
% onutil
1> CREATE DBSLICE orders_sl
2> FROM COGROUP orders_cogroup
3> CHUNK "/dev/dbsl_orders.%r(1..3)";
```

The database server creates the following dbspaces on the eight coservers:

```
coserver   dbspace_identifier  primary chunk
xps.1      orders_sl.1         /dev/dbsl_orders.1
xps.1      orders_sl.2         /dev/dbsl_orders.2
xps.1      orders_sl.3         /dev/dbsl_orders.3
xps.2      orders_sl.4         /dev/dbsl_orders.1
xps.2      orders_sl.5         /dev/dbsl_orders.2
xps.2      orders_sl.6         /dev/dbsl_orders.3
. . .
xps.8      orders_sl.22        /dev/dbsl_orders.1
xps.8      orders_sl.23        /dev/dbsl_orders.2
xps.8      orders_sl.24        /dev/dbsl_orders.3
```

Creating Dbslices for Temporary Files

In addition to dbslices that you create for tables and logical logs, create one or more dbslices for the temporary output that is part of DSS query processing.

Create the dbslices for temporary files across all coservers for balanced use of resources on coservers. Specify these dbslices in either the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable. For more information, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

If temporary dbspaces are evenly distributed across coservers, the database server can use collocated joins to build and process temporary files locally. Other temporary activity is distributed by round-robin across all temporary dbslices.

You can also place explicit temporary tables in the temporary dbspaces.

For information on fragmentation for temporary files, refer to "Creating and Specifying Dbspaces for Temporary Tables and Sort Files" on page 9-60.

Designing a Distribution Scheme

After you decide whether to fragment table rows, index keys, or both, and the dbslices and dbspaces in which the table fragments are distributed, you decide on a scheme to implement this distribution.

The database server supports the following distribution schemes:

- **Round-robin.** This distribution scheme places rows as they are inserted one after another in fragments, rotating through the series of fragments to distribute the rows evenly.

  As rows are added by INSERT statements, the database server uses a hash function on a random number to determine the fragment in which to place the row. If rows are added by INSERT cursors, the database server places the first row in a random fragment, the second in the next fragment sequentially, and so on. If one of the fragments is full, it is skipped.

- **System-defined hash.** This distribution scheme uses an internal, system-defined rule that distributes rows with the intent of keeping approximately the same number of rows in each fragment.

  One advantage of system-defined hash distribution over round-robin distribution is that the database server can identify the fragment in which a row is placed, so that table fragments can be eliminated if they are not needed for query or transaction processing.

- **Expression-based.** This distribution scheme puts rows that contain specified values in the same fragment. You specify a *fragmentation expression* that defines criteria for assigning rows to each fragment, either as a range rule or some other arbitrary rule. Although you can specify a *remainder fragment* that holds all rows that do not match the criteria for any other fragment, a remainder fragment reduces the efficiency of the expression-based distribution scheme.

- **Hybrid fragmentation**. This distribution scheme combines two fragmentation strategies on the same table for increased data granularity and fragment elimination during query processing.

  Hybrid fragmentation provides a two-dimensional fragmentation scheme such that a table is usually fragmented by expression into specific dbslices or sets of dbspaces and fragmented by hash into the specific dbspaces in each dbslice or set of dbspaces.

  You can also use range fragmentation to create a hybrid range fragmentation scheme that uses ranges and subsets of the ranges to divide a table into smaller fragments. For more information about hybrid range fragmentation, see "Creating a Range Distribution Scheme" on page 9-36.

- **Range fragmentation.** This distribution scheme can be used either as a single-level scheme or a hybrid scheme. Range fragmentation is similar to clustering. It is designed to distribute and cluster rows evenly, and it improves access performance for tables with dense, uniform distributions and little or no duplication in the fragmentation column.

This manual emphasizes distribution schemes that have general performance advantages. For a description of possible variations on these distribution schemes, refer to the *Informix Guide to Database Design and Implementation*.

Choosing a Distribution Scheme

The following table compares expression-based, hybrid, round-robin, range, and system-defined hash distribution schemes for three important features: ease of data balancing, fragment elimination, and data skip.

Distribution Scheme	Ease of Data Balancing	Fragment Elimination	Data Skip
Expression-based	You must know the data distribution to balance the number of rows in fragments.	If expressions on one or two columns are used, the database server can eliminate fragments for queries that have either range or equality expressions. Fragmentation expressions that use functions such as DAY(date_field) do not provide fragment elimination unless the query filter uses an expression of the form `column = "literal"`, such as `date_field = "01-04-98"`. The literal expression can also be a host variable or an SPL routine variable.	You can determine whether the integrity of a transaction has been compromised when you use the dataskip feature. You cannot insert rows if the fragment for those rows is not available.
Round-robin	The database server automatically balances data over time.	The database server cannot eliminate fragments.	You cannot determine if the integrity of the transaction is compromised when you use the dataskip feature. You can insert into a table that is fragmented by round-robin if any table fragment is accessible, even though some fragments are not available.
System-defined hash	The database server does not necessarily balance data over time. If the fragmentation column contains unique, well-distributed values, data might be balanced.	The database server can eliminate fragments for queries that have equality expressions.	You cannot determine if the integrity of the transaction is compromised when you use the dataskip feature. You cannot insert rows if the fragment for those rows is not available.

(1 of 2)

Distribution Scheme	Ease of Data Balancing	Fragment Elimination	Data Skip
Hybrid	You must know the data distribution and queries run against the table to balance the table I/O (lookups) across the dbslice. Unless the hash fragmentation column contains evenly distributed unique values, hash-fragmented tables do not necessarily balance data.	The database server can eliminate fragments for range or equality statements on the base-level expression strategy or equality expressions on the secondary-level hash strategy.	You can determine whether the integrity of a transaction has been compromised when you use the dataskip feature. You cannot insert rows if the fragment for those rows is not available.
Range	Data balancing requires dense, uniform, non-duplicate distribution of fragmentation column data.	The database server can eliminate fragments for queries with equality and range expressions.	You can determine whether the integrity of a transaction has been compromised when you use the dataskip feature. You cannot insert rows if the fragment for those rows is not available.

(2 of 2)

The following factors should determine the distribution scheme that you choose, as the previous table describes:

- Whether your queries tend to scan the entire table
- Whether you know the distribution of data to be added
- Whether your applications tend to delete many rows
- Whether you cycle your data through the table

Round-robin distribution schemes are certain to balance data. However, with round-robin distribution, the database server has no information about which fragment contains a specific row, and it cannot eliminate fragments. With system-defined hash distribution, the database server can determine which fragment contains a row, so it *can* eliminate fragments.

In general, choose round-robin or system-defined hash fragmentation only when all the following conditions apply:

- Your queries tend to scan the entire table.
- You do not know the distribution of data to be added.
- Your applications tend not to delete many rows. (The deletion of many rows can degrade load balancing.)

Choose an expression-based, range, or hybrid distribution scheme to fragment the data if any of the following conditions apply:

- Many decision-support queries scan specific portions of the table.
- You know what the data distribution is.
- You plan to cycle data through a database by adding and removing fragments.

You might also use a hybrid distribution scheme to fragment the data if any of the following conditions apply:

- The size of the data is so large that you can improve fragment elimination and parallel processing by further fragmenting each expression-based fragment into two or more hash-based fragments.
- To reduce contention among numerous transactions or decision-support queries that scan specific portions of the table, your application requires a finer granularity of fragments than a simple expression-based or range scheme provides.

Choose a range distribution scheme if both of the following conditions apply:

- The column used for fragmentation is densely and uniformly distributed in the table.
- The fragmentation column contains few or no duplicate values.

In some cases, an appropriate index scheme can circumvent the performance problems of a particular distribution scheme. For more information, refer to "Fragmenting Indexes" on page 9-52.

If you plan to add and delete large amounts of data periodically based on the value of a column such as date, use that column in the distribution scheme. You can then use the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements to cycle the data through the table.

The ALTER FRAGMENT ATTACH and DETACH statements provide the following advantages over bulk loads and deletes:

- The rest of the table fragments are available for access. Only the fragment that you attach or detach is not available.
- The execution of an ALTER FRAGMENT ATTACH or DETACH statement is much faster than a bulk load or mass delete.

For more information, refer to "Attaching and Detaching Table Fragments" on page 9-62.

Choosing a Distribution Scheme for DSS Applications

As you consider possible distribution schemes for DSS applications, keep the following factors in mind:

- If queries scan an entire table, fragment the table so that one or two fragments for each physical CPU are available to the coserver. If each node hosts more than one coserver, divide the number of CPUs available on the node by the number of coservers.
- For optimal performance in decision-support queries, fragment the table to increase parallelism but do not fragment the indexes. Create detached indexes in a separate dbspace.
- Use system-defined hash fragmentation on data when decision-support queries scan the entire table. System-defined hash fragmentation is a good way to spread data evenly across disks and provides the following advantages over the round-robin distribution scheme:
  - You can eliminate fragments for equality expressions on the hash column.
  - You can take advantage of collocated joins when you join on the hash column.

Choosing a Distribution Scheme for OLTP Applications

As you consider distribution schemes for OLTP applications, keep the following factors in mind:

- For improved performance in OLTP, fragment indexes to reduce contention between sessions. You can often fragment an index by its key value, which means the OLTP query has to look at only one fragment to find the location of the row.

  If the key value does not reduce contention, such as when every user is using the same set of key values (for instance, a date range), consider fragmenting the index on another value used in the WHERE clause. To reduce fragment administration, consider not fragmenting some indexes, especially if you cannot find a good fragmentation expression to reduce contention.

- If you fragment tables by expression or range, create fragments to balance I/O requests across coservers and disks instead of balancing quantities of data.

  For example, if most transactions access only some of the rows in the table, set up the fragmentation expression to spread active portions of the table across coservers and disks even if this arrangement results in an uneven distribution of rows.

Creating a System-Defined Hash Distribution Scheme

The optimal fragmentation strategy for a DSS-only environment is to fragment tables across all coservers by system-defined hash on a key that is used for joining the tables. When you fragment tables by system-defined hash on one column, make sure that dbslices have been created to take advantage of collocated joins, as the following section describes.

Hybrid fragmentation schemes, described in "Creating a Hybrid Distribution Scheme" on page 9-34, might provide additional performance benefits.

Ensuring Collocated Joins

When you hash on the same column that is used for joins, you can obtain collocated joins. A *collocated join* is a join that occurs locally on the coserver where the data resides. Because the local coserver sends data to the other coservers after the join is completed, less data is sent between coservers.

For information about creating dbslices to take advantage of collocated joins, see page 9-20.

To take advantage of collocated joins

1. Create the dbslices so that the dbspace ordinal numbers are assigned round-robin across coservers.

```
% onutil
  1> CREATE DBSLICE cust_dbsl FROM
  2> COGROUP sales CHUNK "/dbspaces/dbs1%c" SIZE 1024,
  3> COGROUP sales CHUNK "/dbspaces/dbs2%c" SIZE 1024;
```

   Suppose you have five coservers with two disks on each coserver. This **onutil** command creates dbspaces **cust_dbsl.1** through **cust_dbsl.5** on separate coservers and dbspaces **cust_dbsl.6** through **cust_dbsl.10** on separate coservers. Figure 9-2 shows that dbspaces **cust_dbsl.1** and **cust_dbsl.6** are on the first coserver, dbspaces **cust_dbsl.2** and **cust_dbsl.7** are on the second coserver, and so on.

   Create dbslice **order_dbsl** to distribute dbspaces across the same coservers in the same way.

2. Fragment the tables by hash on the same column that is used to join in the query.

```
CREATE TABLE customer (cust_id integer, ...)
   FRAGMENT BY HASH (cust_id)  IN cust_dbsl;
CREATE TABLE order (... cust_id integer, ...)
   FRAGMENT BY HASH (cust_id)  IN order_dbsl;
```

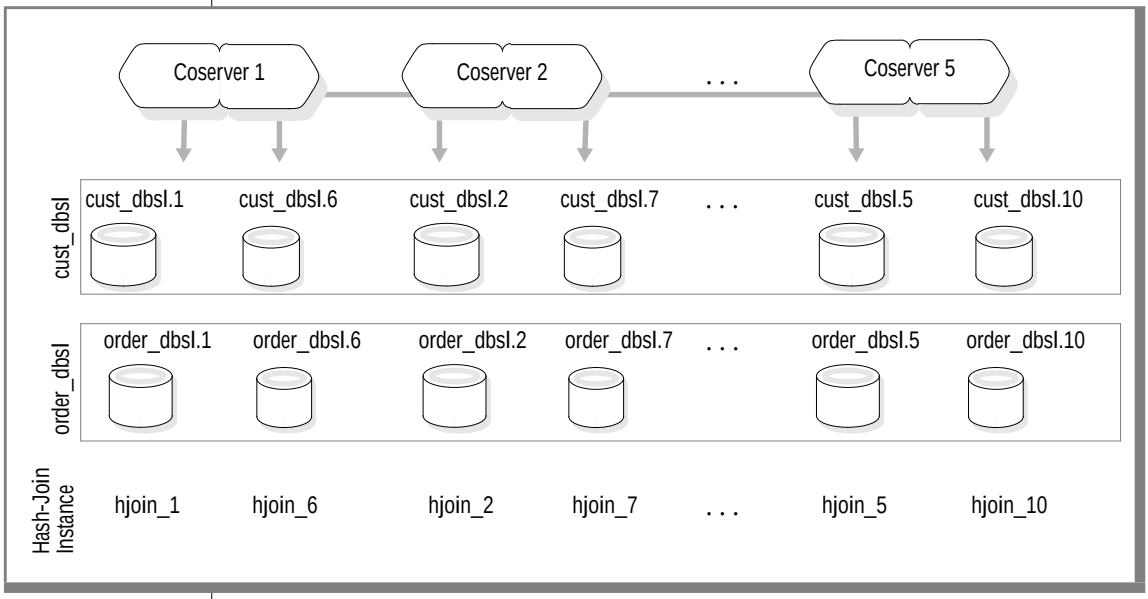
3. Execute the following query:

```
SELECT ... FROM customer c, order o
   WHERE c.cust_id = o.cust_id ...
```

Figure 9-2 shows that the hash join instances for this query match the dbspace ordinal numbers of the dbslice.

Output from the **onstat -g ath** command shows the hash-join instances. Sample **onstat -g ath** output appears in Figure 12-18 on page 12-45. For more information on using the **onstat** command, refer to "Using Command-Line Utilities to Monitor Queries" on page 12-30.

Figure 9-2
Dbspace Ordinal Numbers and Hash-Join Instances for Collocated Joins



Fragmenting on a Serial Column

You can specify a serial column in the system-defined hash distribution scheme of a fragmented table. If you specify a serial column, it must be the only column that you specify in the system-defined hash distribution scheme.

You might notice a difference in serial values that are assigned to the serial column in a fragmented table and a nonfragmented table. The database server assigns serial values in each fragment. However, the values are do not overlap. You cannot specify what values to use. The database server controls the values to add. For more information about the FRAGMENT BY clause, refer to the *Informix Guide to SQL: Syntax*.

Tip: *Although you can create your own hash algorithm, if you use the system-defined hash algorithm for queries and fragmentation, the database server can often eliminate fragments from query processing.*

Creating an Expression-Based Distribution Scheme

The first step in creating an expression-based distribution scheme is to determine the distribution of data in the table, particularly the distribution of values for the column on which you base the fragmentation expression.

To obtain the distribution of values for the fragmentation column

1. Run the UPDATE STATISTICS statement in MEDIUM or HIGH mode for the table and specify the column.
2. Use the **dbschema** utility to examine the distribution.

   For examples of **dbschema** use and output, see the *Informix Migration Guide*.

If you know the data distribution, you can design a fragmentation rule that distributes data across dbspaces as required to meet your fragmentation goal. If your primary goal is to improve performance for DSS queries, a fragment expression that generates a relatively even distribution of rows across fragments might improve parallel processing of table fragments if all fragments are required by most queries.

For fragment elimination, the fragmentation scheme should match most query filters. For example, a table might be fragmented so that data for the days of the month, such as the first, the second, the third, and so on, are stored in separate fragments. However, if queries require a range of data by month, such as the first of January through the thirty-first of March, fragment elimination cannot occur. The fragmentation expression must be a simple expression that uses a range of dates, such as the following example:

```
...
FRAGMENT BY EXPRESSION
   bill_date >= "01/01/1998" AND bill_date < "02/01/1998" IN
dbsp2,
   bill_date >= "02/01/1998" AND bill_date < "03/01/1998" IN
dbsp3,
...
```

The table on page 9-25 describes the limitations of date expressions.

If your primary fragmentation goal is improved concurrency for OLTP applications, analyze the queries that access the table:

- If certain rows are accessed more often than others, try to distribute data so that these rows are not in the same table fragment.
- Avoid specifying columns in the fragmentation expression if they are updated often. Such updates might cause rows to move, deleting them from one fragment and adding them to another fragment. This activity increases CPU and I/O overhead.
- If all columns are used in many transactions, fragmentation expressions based on more than one column can improve fragment elimination. For example, you might create a hybrid fragmentation scheme, fragmenting the table across coservers with an expression-based scheme and across dbspaces within each coserver with a hash-based scheme.

  You might also consider one of the following specialized types of expression-based distribution schemes:

  - Overlapping or noncontiguous fragments based on one or two columns
  - Nonoverlapping fragments based on one or two columns
  - A REMAINDER fragment

These specialized expression-based distribution schemes are not recommended because they do not permit fragment elimination. For information about these modifications of expression-based distribution schemes, see the *Informix Guide to Database Design and Implementation*.

When the query optimizer determines that the values that the WHERE clause selects do not reside on certain fragments, the database server can eliminate those fragments from query processing. For more information, refer to "Designing Distribution for Fragment Elimination" on page 9-41.

Creating a Hybrid Distribution Scheme

Hybrid fragmentation combines system-defined hash and expression-based distribution schemes to increase parallel access and fragment elimination.

Hybrid fragmentation provides a two-dimensional fragmentation scheme such that a table is fragmented by expression into specific dbslices on one dimension and into the dbspaces that the dbslice contains on another dimension. This distribution strategy provides finer granularity of table fragments and permits the database server to eliminate fragments based on the hash distribution, the expression distribution, or occasionally both.

For even finer granularity on a single column, you can use the same column as the distribution column for both the hash and expression distribution.

As an example, suppose that you have created dbslices defined across 12 coservers, and the dbslice includes two dbspaces on each coserver for a total of 24 dbspaces, as the following **onutil** commands show:

```
% onutil
1> CREATE COGROUP sales
2> FROM xps42t_techpubs.%r(1..12);
Cogroup successfully added.
3> CREATE DBSLICE acct_dbsl FROM
4> COGROUP sales CHUNK "/dbspaces/dbs1.%c" SIZE 490000,
5> COGROUP sales CHUNK "/dbspaces/dbs2.%c" SIZE 490000;
Dbslice successfully added.
...
36> CREATE DBSLICE acct_dbsl12 FROM
37> COGROUP sales CHUNK "/dbspaces/dbs23.%c" SIZE 490000,
38> COGROUP sales CHUNK "/dbspaces/dbs24.%c" SIZE 490000;
Dbslice successfully added.
```

For more information on how to create a cogroup and a dbslice, refer to the *Administrator's Guide*.

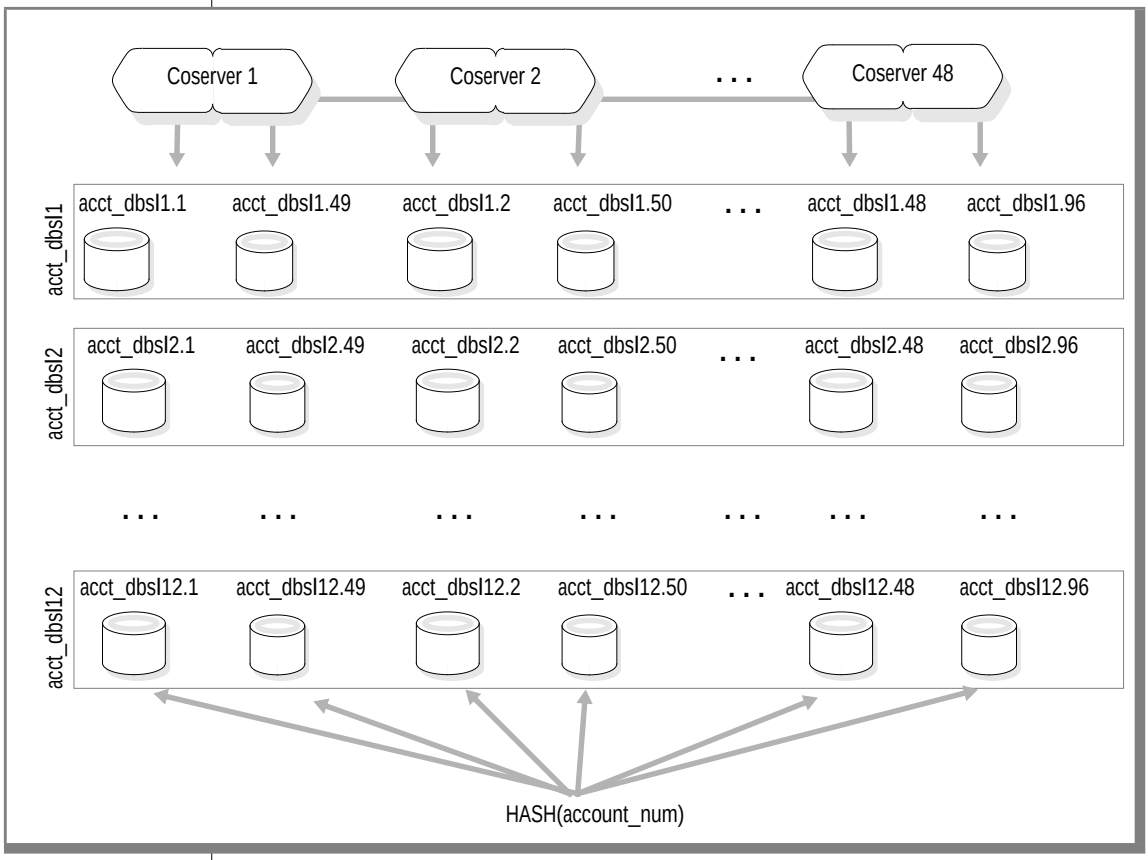
If you have accounting data for a full year, you might fragment account numbers by month, with each month in a separate dbslice. The following CREATE TABLE statement in SQL shows the combination of the expression-based distribution scheme to fragment each month into one of the twelve dbslices and the hash distribution scheme to fragment the monthly account numbers into the dbspaces in each dbslice:

```
CREATE TABLE account
(account_num integer,
account_bal integer,
account_date date,
account_name char(30)
)
FRAGMENT BY HYBRID (account_num) EXPRESSION
   account_date >= '01/01/1996'
      and account_date < '02/01/1996' IN acct_dbsl1
   account_date >= '02/01/1996'
      and account_date < '03/01/96' IN acct_dbsl2
   ...
   account_date >= '12/01/1996'
      and account_date < '01/01/97' IN acct_dbsl12
```

For more information on hybrid fragmentation syntax, see the *Informix Guide to SQL: Syntax*.

Figure 9-3 illustrates the dbspaces in each dbslice for the table fragments that this hybrid distribution scheme defines.

Figure 9-3
Hybrid Fragmentation



Creating a Range Distribution Scheme

Range distribution ensures that rows are fragmented evenly across dbspaces and the fragment that contains each row is uniquely identified. Only columns that contain data of type INTEGER or SMALLINT can be used for range fragmentation expressions, and simple fragmentation can be based on only one column.

Hybrid range fragmentation schemes can specify different columns in each fragmentation statement. For small number ranges, a hybrid range fragmentation scheme might avoid data skew that sometimes occurs with hybrid hash fragmentation schemes.

Range distribution is similar to expression distribution in that rows are distributed by a range of values. You can use range fragmentation in any circumstances in which expression fragmentation is appropriate and the fragmentation columns are INTEGER or SMALLINT data types.

In range distribution, however, the database server balances the distribution of rows evenly among fragments on the basis of the MIN and MAX values if they are provided or on the assumption that the single stated RANGE value is the maximum and `0` is the minimum.

Equality searches on the search key or keys are faster when rows are grouped in range partitions. For example, queries and transactions with filters of the form `WHERE a.col1 = b.col1` or `WHERE a.col1 = '12345'` can take advantage of the range function on `col1` if either table **a** or **b** is a range-fragmented table.

The following simple example shows how to create a table with the RANGE fragmentation option. This example shows how to fragment an account lookup table evenly across the ten dbspaces in the dbslice **accounts** so that each dbspace contains approximately 900 rows. The MIN and MAX keywords indicate the total range of expected values, with account numbers beginning at 1000 and ending at 9999.

```
CREATE TABLE accth(account_num integer,...)
   ...
   FRAGMENT BY RANGE (account_num MIN 1000 MAX 9999) IN accounts;
```

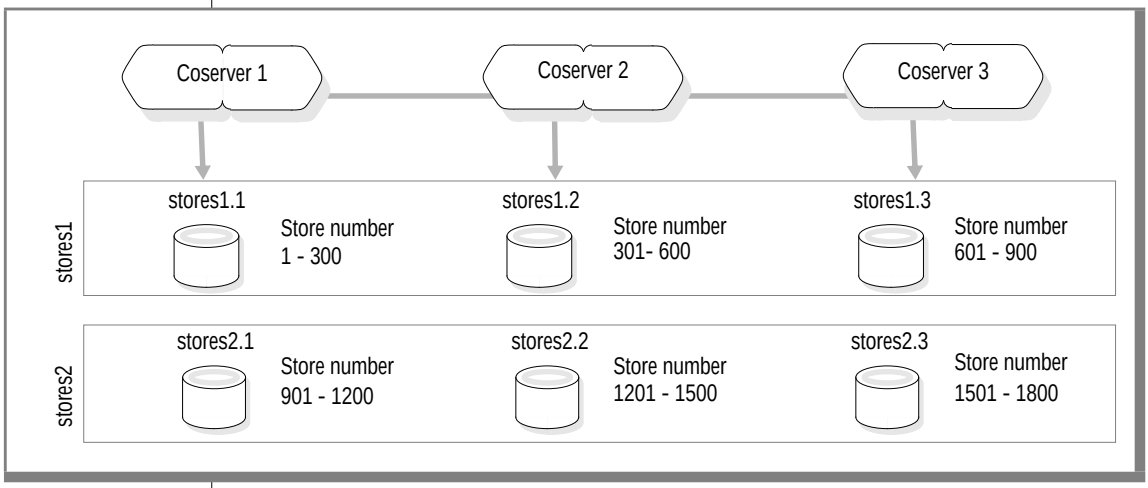
Range fragmentation can be used in a hybrid distribution scheme if the RANGE keyword is used for both fragmentation statements.

For example, assuming that store numbers are evenly distributed from 1 to 1800 and no store number can be greater than 1800, you might enter the following statement to fragment a file on one column in dbslices across coservers and on the same column through dbspaces in the dbslices on each coserver:

```
CREATE TABLE stores(store_num integer,...)
   ...
   FRAGMENT BY HYBRID (RANGE (store_num))
   RANGE(store_num MIN 1 MAX 1800)
IN stores1, stores2
```

Figure 9-4 shows how the store numbers will be distributed among the dbspaces in the **stores1** and **stores2** dbslices.

Figure 9-4
Range Fragmentation



If any values of **store_num** fall outside of the specified range, 0 to 1800, the database server returns an error and does not insert the rows that contain those values into the table. To prevent such problems, you can specify a REMAINDER fragment in a single dbspace. However, rows in a REMAINDER fragment reduce the efficiency of a range-fragmented table for queries that require range searches.

For finer granularity, you can also use range fragmentation to create a hybrid range fragmentation scheme on two columns. For detailed information about the rules for hybrid range fragmentation, refer to the *Informix Guide to SQL: Syntax*.

Although the ALTER FRAGMENT... INIT statement can be used to fragment an existing nonfragmented table with range fragmentation, the ALTER FRAGMENT statements ATTACH or DETACH and ADD, DROP, or MODIFY are not supported for range-fragmented tables.

For detailed information about the syntax of hybrid range fragmentation expressions, see the *Informix Guide to SQL: Syntax*.

Altering a Fragmentation Scheme

If you find that the original fragmentation scheme is not efficient, you can change it with the SQL statement ALTER FRAGMENT, except for range-fragmented tables. Use ALTER FRAGMENT with the INIT keyword to re-create the table with a different fragmentation strategy.

To add a dbspace to a dbslice, use the **onutil** ALTER DBSLICE statement. After you add the dbspace, use the ALTER FRAGMENT statement with the INIT keyword to refragment the table so that it uses the new dbspace.

For more information about the ALTER FRAGMENT statement, refer to the *Informix Guide to SQL: Syntax*. For information about the **onutil** ALTER DBSLICE command, refer to the *Administrator's Reference*.

General Fragmentation Notes and Suggestions

Use the following suggestions as guidelines to fragment tables and indexes:

- Keep fragmentation expressions simple. Although fragmentation expressions can be as complex as you want, complex expressions take more time to evaluate and usually prevent fragment elimination for queries.
- Avoid expressions that require a data type conversion. These conversions increase the time to evaluate the expression. For example, a DATE data type is implicitly converted to INTEGER for comparison purposes.

- Do not fragment tables by expression on columns that contain a value that is used to remove rows from the table unless you are willing to incur the administration costs.

  For example, if you fragment a table on a date column and then delete rows that contain older dates to keep the table up-to-date, the fragments with the oldest dates eventually become empty and the fragments with the recent dates overfill unless you use the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements to add and remove table fragments.
- If you fragment indexes in the same way as the table, performance improves for the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements. For more information, refer to "Attaching and Detaching Table Fragments" on page 9-62.
- Do not fragment every table:
  - Identify the large or critical tables that are accessed most frequently and focus on fragmenting them efficiently.
  - Do not fragment small tables. A small table is a table that occupies one extent or less.
- Create the number of fragments on a coserver as a multiple of the number of its processors and its disks.
- Make sure that you do not allocate too much disk space for each fragment. When you fragment an unfragmented table, make sure that the next-extent size is appropriate for the expected growth of the table fragment.

Designing Distribution for Fragment Elimination

Fragment elimination is a database server feature that reduces the number of table fragments that the database server must access for a query or transaction. Eliminating fragment access can improve performance significantly in the following ways:

- Reduces contention for the disks on which fragments reside

  In some cases, all of the fragments on a given coserver can be eliminated. In other cases, all of the fragments in a given dbslice can be eliminated.

- Improves both response time for a given query and concurrency between queries

  Because the database server does not need to scan unnecessary fragments, I/O for a query is reduced.

- Reduces activity in the LRU queues

  The database server does not need to scan unnecessary fragments into the buffer pool.

If you use an appropriate distribution scheme, the database server can eliminate fragments from the following database operations:

- The fetch portion of the SELECT, INSERT, DELETE or UPDATE statements in SQL

  The database server can eliminate fragments when these SQL statements are optimized, before the actual search.

- Nested-loop joins

  When the database server obtains the key value from the outer table, it can eliminate fragments to search on the inner table.

Depending on the form of the query expression and the distribution scheme of the table, the database server can eliminate either individual fragments or sets of fragments before it performs the actual search.

Whether the database server can eliminate fragments from a search depends on two factors:

- The form of the query expression (the expression in the WHERE clause of a SELECT, INSERT, DELETE or UPDATE statement)
- The distribution scheme of the table that is being searched

Queries for Fragment Elimination

The filter expression, which is the expression in the WHERE clause, is a primary factor in determining the table fragments that must be accessed to satisfy a query or transaction.

The WHERE clause expression can consist of any of the following expressions:

- Simple expression
- Multiple expressions
- Not simple expression

Nevertheless, for fragment elimination, the database server considers only simple expressions or several simple expressions that are combined with certain operators.

A simple expression consists of the following parts:

```
column operator value
```

Simple-Expression Part	Description
column	A single column name Extended Parallel Server supports fragment elimination on all column types except columns that are defined on the TEXT and BYTE data types.
operator	An equality or range operator
value	A literal or a host variable

The following examples show simple expressions:

```
name = "Fred"
date < "01/25/1994"
value >= :my_val
```

The database server considers two types of simple expressions for fragment elimination, based on the operator:

- Range expressions
- Equality expressions

The following examples are not simple expressions and cannot be used for fragment elimination:

```
unitcost * count > 4500
price <= avg(price)
result + 3 > :limit
```

Important: *Built-in functions such as DAY(date-col) are considered complex expressions.*

Range Expressions in Query

Range expressions use the following relational operators:

- <
- >
- <=
- >=
- !=

The database server can eliminate fragments for queries that contain any combination of these relational operators with one or two of the columns used for fragmentation in the WHERE clause.

The database server can also eliminate fragments when these range expressions are combined with the following operators:

- AND
- OR
- NOT
- IS NULL
- IS NOT NULL

If the range expression contains MATCHES or LIKE, the database server can also eliminate fragments if the string ends with a wildcard character. However, if the expression involves an NCHAR or NVARCHAR column with MATCHES or LIKE, the database server cannot eliminate fragments.

The following examples show query expressions that can take advantage of fragment elimination:

```
columna MATCHES "ab*"
columna LIKE "ab%" OR columnb LIKE "ab_"
```

Equality Expressions in Query

Equality expressions use the following equality operators:

- =
- IN

The database server can eliminate fragments for queries that contain any combination of these equality operators with one or two of the columns used for fragmentation in the WHERE clause.

The database server can also eliminate fragments when these equality expressions are combined with the following operators:

- AND
- OR

Types of Fragment Elimination

The database server provides the following types of fragment elimination:

- **Range Elimination.** This type of fragment elimination determines which fragments to exclude from the scan based upon an appropriate combination of:
  - a range-based distribution scheme or an expression-based distribution scheme in which fragments are ordered so that overlaps in the expressions do not result in any ambiguity about which fragment contains a particular fragmentation key.
  - a query expression that contains range or equality expressions on values of the columns specified in the range-based or expression-based distribution scheme.
- **Hash Elimination.** This type of fragment elimination determines which fragments to exclude from the scan based upon an appropriate combination of:
  - a hash-based distribution scheme.
  - a query expression that contains equality expressions that involve values of the columns specified in the hash-based distribution scheme.

Range Elimination

For simple expressions, range elimination can occur if the column is used for range or expression-based table fragmentation. If the column is not used for table fragmentation, range elimination cannot occur for the expression.

Range elimination can occur only for simple expressions. "Queries for Fragment Elimination" on page 9-42 describes simple expressions.

If expressions for which fragment elimination can occur are connected by either AND or OR operators, range elimination can occur for the resulting combined expression. For example, range elimination can occur if the columns specified in the WHERE clause of the query are all columns used for table fragmentation.

```
name = "Fred" AND date < "01/25/1994"
region IN (1,2,3,4) AND sales > :min
cost < 100.00 OR quantity < 5
```

If an expression for which range elimination cannot occur is connected to an expression for which range elimination can occur by an AND operator, range elimination can occur for the resulting combined expression.

If an expression for which range elimination cannot occur is connected to an expression for which range elimination can occur by an OR operator, range elimination cannot occur for the resulting combined expression.

In the following examples, assume that **region** is a column used for fragmentation and **sales** is not. Range elimination can occur for the following expressions:

```
region IN (1,2,3,4) AND sales > 10000
(region = 1 AND sales > 10000) OR (region = 2 AND sales < 100)
```

Range elimination cannot occur for the following examples:

```
region IN (1,2,3,4) OR sales > 10000
(region = 1 AND sales > 10000) OR sales < 100
```

Hash Elimination

For simple expressions, hash elimination can occur if the column is used for hash fragmentation of the table. If the column is not used to fragment the table, hash elimination cannot occur for the expression.

For example, suppose you define hash fragmentation on column **account_num**. The following examples show simple expressions for which hash elimination can occur:

```
account_num = 12345
account_num IN (12345, 11111, 23987)
```

Hash elimination cannot occur for expressions that do not fit the preceding definition of simple expressions or do not use an equality operator. For example, hash elimination cannot occur for the following expressions:

```
price = AVG(price)
orderno > 1000
```

All columns in the hash distribution specification must be present in a simple equality expression, and all such expressions must be connected with the AND operator. If not all hash columns are present in an expression, or any expression on a hash column is not logically connected to the others by an AND operator, no hash elimination can be performed for the overall expression.

For example, if you define hash fragmentation on columns **a** and **b**, hash elimination can occur for the following expressions:

```
a = 1 AND b = 2
a = 1 AND b = 2 AND color = "Red"
a IN (1,2,3) AND (b = 4 OR b = 5)
```

Hash elimination cannot occur for the following expressions:

```
a = 1 OR b = 2
(a = 1 AND b = 2) OR color = "Red"
a IN (1,2,3) OR (b = 4 OR b = 5)
```

An expression for which hash elimination can occur can be combined with other such expressions using either AND or OR operators, and hash elimination can occur for the resulting expression.

If an AND operator connects an expression for which hash elimination cannot occur to an expression for which hash elimination can occur, hash elimination can occur for the resulting combined expression. For example, suppose you define hash fragmentation on column **orderno**. Hash elimination can occur for the following query expressions:

```
orderno = 2001 OR orderno = 2002
orderno IN (2001,2002,4095) AND color = "Red"
orderno = 2001 OR (color = "Red" AND orderno = 2002)
```

Hash elimination cannot occur for the resulting combined expression if an OR operator connects an expression for which hash elimination cannot occur to an expression for which hash elimination can occur.

For example, hash elimination cannot occur for the following expression:

```
orderno IN (2001,2002) OR color = "Red"
```

Query and Distribution Scheme Combinations for Fragment Elimination

Figure 9-5 summarizes the fragment-elimination behavior for Extended Parallel Server. It also shows the different combinations of distribution schemes and query expressions that involve the following columns:

- Fragmentation columns

  A fragmentation column is a column that you specify in the distribution scheme when you define the fragmentation strategy
- Nonfragmentation columns that are connected by the AND operator to an expression for which fragment elimination can occur

Figure 9-5
Fragment-Elimination Behavior

Distribution Scheme	Query Expression Contains		
	Any Number of Equality Expressions on Fragmentation Column	Range Expressions on One or More Columns	Other Expressions
Expressions on three or more columns	None	None	None
Expressions on as many as five columns	Range elimination	Range elimination	Range elimination, if combined with AND operator
Range distribution on one or two columns	Range elimination	Range elimination	Range elimination, if combined with AND operator
Hash on one or multiple columns	Hash elimination	None	Hash elimination, if combined with AND operator
Hybrid with expressions on three or more columns	Hash elimination	None	None
Hybrid with expressions on as many as five columns	Hash elimination and range elimination	Range elimination	None

In the following circumstances, fragment elimination cannot occur:

- Some of the hash fragmentation columns do not appear in an equality expression.
- A query expression contains more than five fragmentation columns and at least one expression on a fragmentation column is a range expression.
- A query expression contains one or more nonfragmentation columns that are connected to the other expressions with an OR operator.
- A query expression contains no fragmentation columns.

Figure 9-5 indicates that the query expression in the WHERE clause of the query in question determines the effectiveness of fragment elimination. The following sections provide examples of distribution schemes and query expressions that enable fragment elimination.

System-Defined Hash Distribution Scheme

The following CREATE TABLE statement shows a fragmentation strategy that uses a system-defined hash distribution scheme to fragment a table across multiple dbspaces in a dbslice:

```
CREATE TABLE account
(account_num integer,
account_bal integer,
account_date date,
account_name char(30)
) FRAGMENT BY HASH(account_num)
IN acct_dbslc;
```

The database server cannot eliminate any of the fragments from the search if the WHERE clause includes a range expression such as the following one:

```
account_num >= 11111 AND account_num <= 12345
```

However, the database server can eliminate all but one fragment from the search if the WHERE clause includes an equality expression, such as the following one:

```
account_num = 12345
```

Hybrid Distribution Scheme

The database server can eliminate fragments from the search based on the hash distribution scheme, the expression-based distribution scheme, or both schemes.

Suppose you have a very large table that keeps a history of account activities for each month in a year. Figure 9-6 on page 9-51 provides a sample hybrid distribution scheme and dbspace layout in each dbslice for this table.

The database server can eliminate some of the fragments from the search if the WHERE clause includes any combination of the following expressions:

- An equality expression on the hash column

  ```
  account_num = 12345
  ```

- An equality expression on the column that is used for the expression-based distribution scheme

  ```
  account_date = '01/01/1996'
  ```

- A range expression with the column that is used for the expression-based distribution scheme

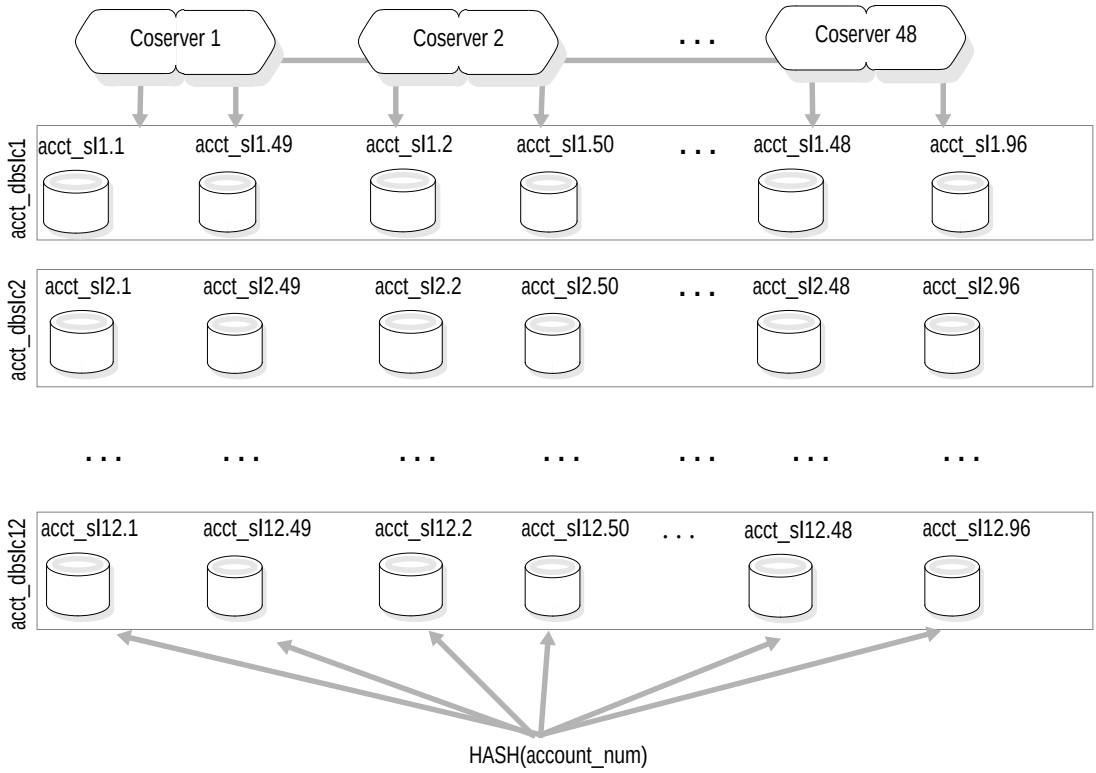
  ```
  account_date >= '01/01/1996'
     AND account_date <='03/01/1996'
  ```

If the WHERE clause in a query includes the following expression, the database server eliminates the fourth through twelfth dbspaces from the search of the hybrid fragmentation layout that Figure 9-6 shows. This query expression reads only three fragments.

```
account_date IN ('01/01/1996','02/01/1996','03/01/1996')
   AND account_num = 12345
```

Figure 9-6
Hybrid Fragmentation and Fragment Elimination

```
FRAGMENT BY HYBRID (HASH (account_num)) EXPRESSION
   account_date < '02/01/1996'
      AND account_date >= '01/01/1996' IN acct_sl1
   account_date < '01/03/1996'
      AND account_date >= '02/01/1996' IN acct_sl2
   ...
   account_date < '01/01/1997'>=
      AND account-date '12/01/1996' IN acct_sl12
```



The database server cannot eliminate fragments defined by the hash distribution scheme if the WHERE clause includes a range expression with this hash column as in the following expression:

```
account_num >= 11111 AND account_num <= 12345
```

Fragmenting Indexes

You choose the fragmentation strategy for your table data. When you create an index on a very large table, the index might also be very large. To improve performance of queries and other database operations, you can also fragment the index across multiple dbspaces.

You can fragment the index with either of the following strategies:

- Same fragmentation strategy as the table (attached index)
- Different fragmentation strategy from the table (detached index)

Attached Indexes

An *attached index* is an index that is created with the same fragmentation strategy as the table. A fragmentation strategy is the distribution scheme and set of dbspaces in which the fragments are located.

Tip: *To avoid rebuilding the entire index for the table, create an attached index if you expect to use ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH to modify the table.*

Because the index is created with the same fragmentation strategy as the table, you do not specify a fragmentation strategy to create an attached index, as the following two sample SQL statements show:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
     (a >= 0 and a < 5) IN dbspace1,
     (a >= 5 and a < 10) IN dbspace2
     ...

CREATE INDEX idx1 ON tb1(a);
```

To fragment the attached index with the same distribution scheme as the table, the database server uses the same rule for index keys as for table data. As a result, attached indexes have the following physical characteristics:

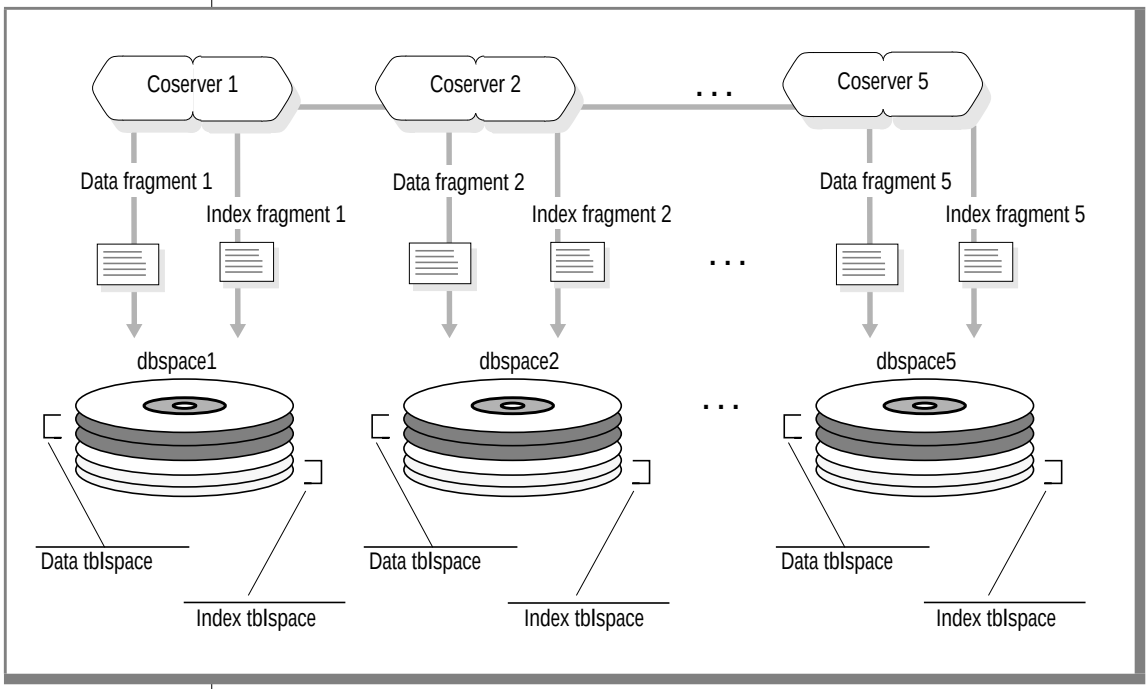
- The number of index fragments is the same as the number of data fragments.
- Each attached index fragment resides in the same dbspace as the corresponding table data, but in a separate tblspace.

  The **partn** column in the **sysfragments** system catalog table contains a different tblspace (partition) number for the index fragment and the table fragment.

Tip: *If you create a unique index on columns that are not used to fragment the table, the index is fragmented by hash into the same dbspaces as the table. The result might be a globally detached index. For information about locally and globally detached indexes, see* *"Choosing an Attached or Detached Index" on page 7-14*.

Figure 9-7 illustrates the storage scheme for the indexes that are attached to a table that is fragmented across five coservers.

Figure 9-7
Storage Scheme for Indexes That Are Attached to a Fragmented Table



Detached Indexes

A common fragmentation strategy is to fragment indexes in the same way as the table but to specify different dbspaces for the index fragments. Putting the index fragments into different dbspaces from the table might improve the performance of operations such as backup and recovery.

Indexes can be detached *locally* on the same coserver with the table or table fragment or *globally*, across coservers.

Detached indexes can be fragmented with any scheme except round-robin. You can specify a hash, expression, or hybrid fragmentation scheme. For information about constraints in indexes, see "Constraints on Indexes for Fragmented Tables" on page 9-56.

If you think that an unfragmented index for a fragmented table might improve performance, place the index in a separate dbspace with the IN *dbspace* clause of the CREATE INDEX statement.

If you decide to create a fragmented detached index for a fragmented table, consider the effect on performance carefully. If an index fragment is on one coserver and its index values are used to look up data in a table fragment on another coserver, the overhead of intercoserver communication slows performance.

A fragmented and globally detached index might be useful for some DSS queries. A globally detached fragmented index might be appropriate in the following cases:

- If queries require singleton selects on columns that are indexed but not used to fragment the table
- If queries require key scans of only the columns contained in the index, particularly if the WHERE clause permits index fragment elimination

The performance impact of table and index updates should be minimal if the table is modified by batch processes that perform large deletes or inserts. However, OLTP transactions that update, add, or delete one or a few rows at a time might not update a globally detached index as quickly as a locally detached or attached index. On the other hand, using a nonfragmentation column to select a single table row might be quicker if the column is used to fragment a detached index.

The following sample SQL statements show how an index might be detached:

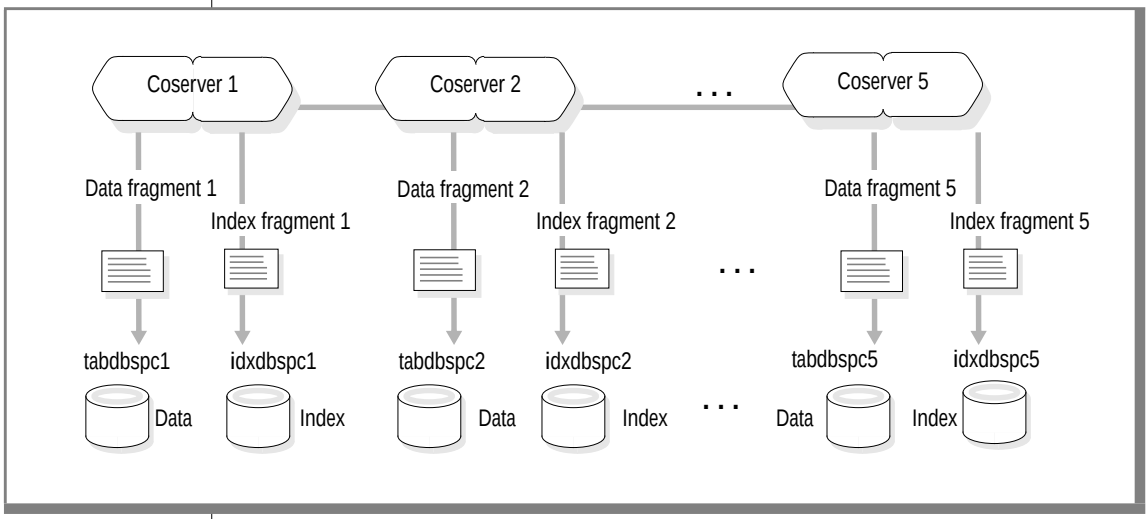
```
CREATE TABLE tb1 (a int)
   FRAGMENT BY EXPRESSION
      (a <= 10) IN tabdbspc1,
      (a <= 20) IN tabdbspc2,
      (a <= 30) IN tabdbspc3;

CREATE INDEX idx1 ON tb1 (a)
   FRAGMENT BY EXPRESSION
      (a <= 10) IN idxdbspc1,
      (a <= 20) IN idxdbspc2,
      (a <= 30) IN idxdbspc3;
```

In this example, the CREATE INDEX statement specifies the same distribution scheme as the table but specifies different dbspaces. Indexes can also be fragmented with a different scheme from the underlying table or globally detached, as described in "Choosing an Attached or Detached Index" on page 7-14.

Figure 9-8 illustrates a possible storage scheme for detached indexes.

Figure 9-8
A Possible Storage Scheme for Detached Indexes



For more information on the CREATE INDEX statement, refer to the *Informix Guide to SQL: Syntax*.

Constraints on Indexes for Fragmented Tables

Unique constraints are permitted on any indexed column, including columns that are not unique keys in the table. If the index is created with constraints, the following strategies apply:

- If the table is not fragmented, the index is placed in the same dbspace with the table.
- If the table is fragmented with some of the keys used to fragment the index, the index is fragmented in the same way as the table.

- If the table is fragmented on different columns from the index, the index is fragmented by hash into the same dbspaces as the table.

  For example, if a table contains columns named **order_num**, **cust_num**, **order_date,** and **ship_meth** and **cust_num** is a foreign key, the table might be fragmented by hash on **order_num**. The fragmented index on **cust_num** for this table is hashed into the same dbspaces as the table. A globally detached index might result, however.

Indexing Strategies for DSS and OLTP Applications

OLTP applications usually access a very small number of rows in a table in a single transaction. Indexes improve the performance of OLTP transactions because only a few index pages and possibly one data page need to be read. Attached indexes, in which the index fragment resides in the same dbspace as the corresponding table data, benefit OLTP queries.

DSS applications usually access many rows of a large table. A DSS query that accesses a large volume of data often performs better if it uses a table scan instead of an index. For optimal performance in decision-support queries, fragment the table to take advantage of fragment elimination and to increase parallelism. Tables and indexes can use different fragmentation strategies and different degrees of fragmentation. Usage patterns might indicate that it is more efficient to separate an index into fewer fragments than its base table and to store the index fragments in different dbspaces or dbslices from the base table.

When a DSS query accesses only values that are part of an index key, the query scans only the index and does not need to access the table. The database server provides the following types of indexes to improve the performance of DSS applications:

- Bitmap indexes

  Bitmap indexes can save disk space if the indexed column contains many duplicate key values, and they can improve performance of queries that use multiple indexes or require counts of duplicated data in the indexed key columns. For information about the performance and space advantages of bitmap indexes, see "Using Bitmap Indexes" on page 13-20.

- Generalized-key (GK) indexes, described in "Using Generalized-Key Indexes" on page 13-23

  A GK index can contain key values that are:

  - a subset of rows from a table.
  - derived from an expression.
  - join columns from multiple tables.
  - a combination of various indexes on a table.

For tables that are not updated, create as many indexes as are useful for queries. The database server can use multiple indexes in a single execution.

For more information on when to use the different types of indexes, refer to "Using Indexes" on page 13-13.

For more information on estimating the amount of space that is required for the different types of indexes, refer to "Estimating Index Page Size" on page 7-6.

Fragmenting Temporary Tables

Just as you fragment permanent tables, you can fragment explicit temporary tables across coservers. The database server provides two kinds of explicit temporary tables. Your choice of one or the other has performance implications.

Scratch tables are nonlogging temporary tables that do not support indexes, constraints, or rollback. *Temp tables* are logged tables although they also support bulk operations such as light appends. Temp tables support indexes, constraints, and rollback. For more information about the characteristics of temporary tables, see "Using Temporary Tables" on page 6-7.

To create a temporary, fragmented table, use the TEMP TABLE or SCRATCH TABLE option of the CREATE TABLE statement. If you use the CREATE TEMP TABLE option, you can specify what fragmentation scheme and which dbspaces to use for the temporary table. The database server fragments SCRATCH tables with the round-robin scheme into the dbspaces or dbslices specified by the DBSPACETEMP configuration parameter or environment variable unless you specify a fragmentation scheme as part of the CREATE TABLE statement.

Temporary tables are subject to the following rules:

- You can define your own fragmentation strategy for an explicit temporary table, or you can let the database server determine the fragmentation strategy dynamically.
- You cannot alter the fragmentation strategy of a temporary table as you can with permanent tables.
- Unless you specify a dbspace or dbslice when you use the CREATE TEMP TABLE statement, the table is stored in the spaces specified by the setting of the ONCONFIG parameter DBSPACETEMP.

For more information about temporary tables, refer to the *Administrator's Guide*. For more information about the SQL keywords that you use to create temporary tables, refer to the *Informix Guide to SQL: Syntax*.

Letting the Database Server Determine the Fragmentation

An explicit temporary table for which the database server determines a fragmentation strategy automatically is called a *flexible (flex) temporary table.*

You do not need to know the column names and data types for flex temporary tables, as you do with temporary tables created with the CREATE TEMP TABLE statement.

The database server creates this kind of table during the execution of the following types of queries:

```
SELECT * FROM customer INTO SCRATCH temp_table
SELECT * FROM customer INTO TEMP temp_table WITH NO LOG
```

If temporary tables created with the INTO TEMP keywords are stored in temporary dbspaces created by **onutil** CREATE TEMP DBSPACE or CREATE TEMP DBSLICE statements, you must specify WITH NO LOG. Tables in temporary spaces are not logged.

When you use SELECT...INTO TEMP syntax, the database server uses a flex temporary table operator to optimize the use of the dbspaces and dbslices as specified by DBSPACETEMP for storage of temporary tables and fragments.

The flex operators execute the insert into these fragments in parallel. The number of CPU VPs available to the query determines the number of parallel instances of the flex SQL insert operator.

If the rows that the query produces are less than or equal to 8 kilobytes, the temporary table resides in only one dbspace. If the rows exceed 8 kilobytes, the temporary table is fragmented with a round-robin distribution scheme. If an instance of the flex insert operator does not receive any data, it does not create any fragments.

Specifying a Fragmentation Strategy

If you use SET EXPLAIN output to analyze query performance, you might see a way to improve performance by specifying the fragmentation for one or more temporary tables. For example, for the advantage of fragment elimination, you might create a temporary table that is fragmented by hash on the same column as another table to be used in a query.

To create a temporary, fragmented table, use the TEMP TABLE option of the CREATE TABLE statement and specify the distribution scheme and dbspaces to use for the temporary table.

For a simple example of an explicitly fragmented temporary table, see "Explicit Temporary Tables" on page 6-8.

Creating and Specifying Dbspaces for Temporary Tables and Sort Files

Because large DSS queries require large amounts of temporary space for sorts and joins, you should provide adequate temporary disk space. Specifying temporary dbspaces prevents queries from putting temporary tables and sort files in the root dbspace or in dbspaces that permanent tables use.

You can define one or more dbslices to distribute temporary space across coservers that you specify, or you can define dbspaces on coservers one at a time. List the dbspaces or dbslices as arguments to the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable. Applications can use the **DBSPACETEMP** environment variable to specify lists of dbspaces for temporary tables and sort files.

To create a dbspace for the exclusive use of temporary tables and sort files, use one of the following commands:

- The **onutil** CREATE TEMP DBSLICE command to create temporary dbspaces across multiple coservers
- The **onutil** CREATE TEMP DBSPACE command to create a temporary dbspace on one coserver

Tip: *Dbspaces and dbslices that you create with the CREATE TEMP DBSPACE or CREATE TEMP DBSLICE command can contain only nonlogging temporary files. If you create logging dbspaces or dbslices and specify these dbspaces or dbslices as arguments to the DBSPACETEMP configuration parameter, you can create logged temporary tables with the INTO TEMP keywords. Implicit temporary files and files created with the INTO SCRATCH keywords are not logged even if they are placed in standard dbspaces.*

Regardless of the way in which you create dbspaces for temporary files, make sure that they are evenly balanced across all coservers. If possible, create dbspaces for temporary files on disks that permanent database files do not use.

For more information, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

Figure 9-9 shows how the dbspaces that DBSPACETEMP lists might be distributed across three disks on a single coserver.

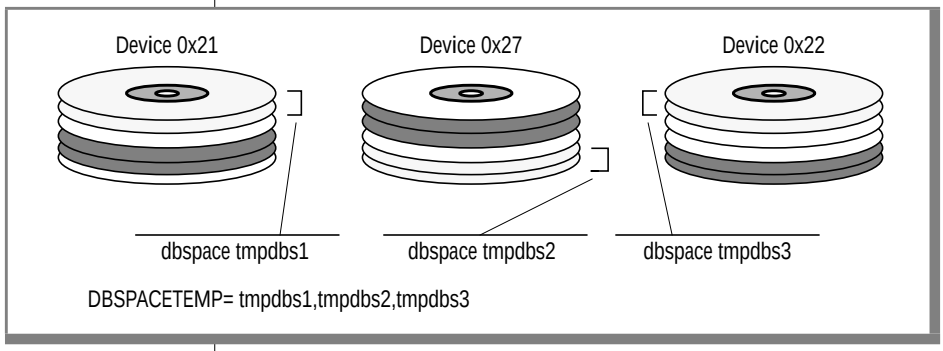


Figure 9-9
Dbspaces for Temporary Tables and Sort Files on a Single Coserver

Attaching and Detaching Table Fragments

Many applications use ALTER FRAGMENT ATTACH and DETACH statements to add or remove a large amount of data in a very large table. ALTER FRAGMENT DETACH provides a way to delete a segment of the table data rapidly. Similarly, ALTER FRAGMENT ATTACH provides a way to load large amounts of data incrementally into an existing table by taking advantage of the fragmentation scheme.

If the database server must completely rebuild detached indexes on the surviving table, the ALTER FRAGMENT ATTACH and DETACH statement can take a very long time to execute. For tables with attached indexes, the database server can reuse the indexes on the surviving or existing fragments and eliminate the index build during the attach or detach operation, with the following results:

- Reduces the time that it takes for the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements to execute
- Improves the table availability

Improving ALTER FRAGMENT ATTACH Performance

To reuse indexes on the surviving table fragments for the ALTER FRAGMENT ATTACH statement, you must meet both of the following requirements:

- Formulate appropriate distribution schemes for your surviving table and index fragments.
- Ensure that the indexes on the attached (consumed) tables are similar to the surviving table.

Formulating Appropriate Distribution Schemes

You reuse existing attached indexes for a table when you execute the ALTER FRAGMENT ATTACH statement. You create an attached index when you create an index without specifying a fragmentation strategy. For more information on attached indexes, refer to “Attached Indexes” on page 9-52.

During execution of the ALTER FRAGMENT ATTACH statement, the database server reuses the existing attached index fragments on the table and builds an index only for the newly attached table fragment.

For example, suppose that you create a fragmented table and attached index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
     (a >= 0 and a < 5) IN db1,
     (a >= 5 and a < 10) IN db2;

CREATE INDEX idx1 ON tb1(a);
```

Now, suppose that you create another table that is not fragmented, and you later decide to attach it to the fragmented table.

```
CREATE TABLE tb2 (a int, check (a >=10 and a<15))
   IN db3;

ALTER FRAGMENT ON TABLE tb1
     ATTACH
       tb2 AS (a >= 10 and a < 15) AFTER db2;
```

This attach operation can take advantage of the existing index **idx1**.The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**. But the database server builds an attached index for just the consumed table, **tb2,** and attaches this index fragment as a fragment of index **idx1.**

During execution of the ALTER FRAGMENT ATTACH statement, the database server rebuilds all detached indexes on the surviving table. For more information on detached indexes, refer to "Detached Indexes" on page 9-54.

Specifying Similar Index Characteristics

The database server might also be able to reuse an existing index on the consumed table if an index on the same columns exists on the surviving table.

To be reused, the index on the consumed table must have the following characteristics:

- It is an index on the same set of columns in the same order as the index on the surviving table.
- It is fragmented in the same way as the consumed table.
- It has the same index characteristics (unique, clustered, and so forth) as the index on the surviving table.

If such an index exists, the database server uses that index and attaches it as a fragment of the corresponding index on the surviving table. If the consumed table does not have such an index, the database server builds only that index fragment on the consumed table.

Suppose you create an index on the consumed table in the previous example with the following SQL statements:

```
CREATE TABLE tb2 (a int, check (a >=10 and a<15))
   IN db3;

CREATE INDEX idx2 ON tb2(a);
```

Then suppose you attach this table to the first table with the following SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
    ATTACH
      tb2 AS (a >= 10 and a < 15) AFTER db2;
```

This attach operation can take advantage of the existing index **idx2**. The database server reuses index **idx2** and converts it into a fragment of index **idx1**.

Improving ALTER FRAGMENT DETACH Performance

To take advantage of the performance optimizations for the ALTER FRAGMENT DETACH statement, you can formulate appropriate distribution schemes for your table and index fragments.

You can eliminate the index build during execution of the ALTER FRAGMENT DETACH statement if you fragment the index in the same way as the table. You fragment an index like the table when you create an index without specifying a fragmentation strategy.

For example, suppose that you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
     (a >= 0 and a < 5) IN db1,
     (a >= 5 and a < 10) IN db2,
     (a >=10 and a < 15) IN db3;
CREATE INDEX idx1 ON tb1(a);
```

The database server fragments the index keys into dbspaces **db1**, **db2**, and **db3** with the same column **a** value ranges as the table because the CREATE INDEX statement does not specify a fragmentation strategy.

Now, suppose that you decide to detach the data in the third fragment with the following SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   DETACH db3 tb3;
```

Because the fragmentation strategy of the index is the same as that for the table, the ALTER FRAGMENT DETACH statement does not rebuild the index on the surviving table after the detach operation. The database server drops the index fragment in dbspace **db3** and updates the system catalogs.

Monitoring Fragmentation

After you fragment tables, it is important to monitor fragment use. Even if you have planned the fragmentation strategy carefully, as applications and queries change and table fragments fill, fragment use patterns might also change.

You can monitor processing balance and fragment use most usefully across coservers, but you might monitor a single coserver if it alone seems to have abnormal use patterns.

Monitoring Fragmentation Across Coservers

You monitor fragment use and I/O request queues across all coservers to find out if I/O queues are unbalanced or if some fragments are used more than others. If you see imbalances, you might adjust the fragmentation strategy.

Use options of the **onstat** command-line utility to monitor the following aspects of fragmentation across all coservers:

- Information about free space in all dbspaces

  This information appears in Figure 9-10 as output of the **xctl onstat -d** command.
- The number of reads and writes to dbspaces, files, and chunks

  This information appears in Figure 9-11 on page 9-68 as output of the **xctl onstat -D** command and in Figure 9-12 on page 9-69 as output of the **xctl onstat -g iof** command.

To display statistics for a coserver to which you are not connected or for multiple coservers, use the **xctl** utility to execute the **onstat** command. The **xctl** prefix to the **onstat** command-line options displays **onstat** information for each coserver that is currently initialized.

xctl onstat -d

The **xctl onstat -d** command displays the following information for each chunk in a dbspace:

- Address of the chunk
- Chunk number and associated dbspace number
- Offset into the device in pages
- Size of the chunk in pages
- Number of free pages in the chunk
- Approximate number of free blobpages
- Path name of the physical device

Figure 9-10 shows an example of the **xctl onstat -d** output for that includes UNIX pathnames.

Figure 9-10
xctl onstat -d Output

```
Dbspaces
address  number   flags      fchunk   nchunks  flags    owner    name
a3a4190  1        0x8000     1        1        N        informix rootdbs.1
 1 active, 32768 maximum (if CONFIGSIZE is LARGE)

Chunks
address  chk/dbs offset(p) size(p)  free(p)  bpages   flags pathname
a3a4288  1   1   0         17500    11967              PO-   /work2/dbfiles/kermit/rootdbs.1
 1 active, 32768 maximum (if CONFIGSIZE is LARGE)

...
Dbspaces
address  number   flags      fchunk   nchDbspaces
address  number   flags      fchunk   nchunks  flags    owner    name
a3a4190  1        0x8000     1        1        N        informix rootdbs.1
 1 active, 32768 maximum (if CONFIGSIZE is LARGE)

Chunks
address  chk/dbs offset(p) size(p)  free(p)  bpages   flags pathname
a3a4288  1   1   0         17500    11967              PO-   /work2/dbfiles/kermit/rootdbs.1
 1 active, 32768 maximum (if CONFIGSIZE is LARGE)
```

For more information about the information that this option displays, refer to the utilities chapter in the *Administrator's Reference*.

xctl onstat -D

The **xctl onstat -D** option displays the same information as **xctl onstat -d** and also the following two fields:

- Number of pages read from the chunk (**page Rd**)
- Number of pages written to the chunk (**page Wr**)

Figure 9-11 shows sample output from the **xctl onstat -D** option**.**

```
...

Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
a10d5a0  1        1        1        1        N        informix rootdbs1
a10d628  3        1        3        1        N        informix customer_dbsl.1
a10d6b0  5        1        5        1        N        informix period_dbsl.1
a10d738  7        1        7        1        N        informix product_dbsl.1
4 active, 100 total

Chunks
address  chk/dbs offset   page Rd  page Wr  pathname
a108c30  1   1   0        74       12       /dev/rdsk/c0t1d0s6
a108d60  3   3   0        53       0        /dev/custdbsl.1
a108e90  5   5   50000    3        0        /dev/perdbsl.1
a108fc0  7   7   55000    481      0        /dev/proddbsl.1
4 active, 100 total
```

Figure 9-11
xctl onstat -D Output

The **page Rd** field in this output shows that chunk 7 has a disproportionately higher number of page reads (481) compared to the other dbspaces.

To determine the dbspace name

1. In the **Chunks** section, find the dbspace number in the **dbs** field (7).
2. Find the same value in the **number** field of the **Dbspaces** section of this output.
3. The **name** field in the **Dbspaces** section tells you the dbspace name (**product_dbsl.1).**

To distribute the frequently accessed rows evenly to other dbspaces, you might change the fragmentation strategy of the tables in the **product_dbsl.1** dbspace.

xctl onstat -g iof

The **xctl onstat -g iof** option displays the number of reads from and writes to each dbspace or file. This option is similar to the **-D** option, except that it also displays information on nonchunk files. It includes information about temporary files and sort-work files.

Use this option to monitor the distribution of I/O requests for the different fragments of a table. If one chunk has a disproportionate amount of I/O activity, it might be a system bottleneck.

The sample output in Figure 9-12 shows that the disk reads and disk writes are not balanced across the different dbspaces.

```
...

AIO global files:
gfd pathname          totalops   dskread   dskwrite  io/s
  3 rootdbs1           86        74        12        0.2
  4 /dev/custdbsl.1    53        53        0         0.1
  5 /dev/perdbsl.1     3         3         0         0.0
  6 /dev/proddbsl.1    481       481       0         0.9
  7 /dev/dbsl.1        89        89        0         0.2

...

AIO global files:
gfd pathname          totalops   dskread  dskwrite   io/s
  3 rootdbs2           510       32        478       1.0
  4 /dev/custdbsl.2    30        30        0         0.1
  5 /dev/perdbsl.2     3         3         0         0.0
  6 /dev/proddbsl.2    483       483       0         0.9
  7 /dev/dbsl.2        89        89        0         0.2
```

Figure 9-12
xctl onstat -g iof Output

Figure 9-12 shows the following data skews:

- The chunks whose pathname is **/dev/proddbsl.1** on the first coserver and **/dev/proddbsl.2** on the second coserver have a disproportionately high number of disk reads compared to the other chunks.

  To distribute the frequently accessed rows evenly to other dbspaces, you might change the fragmentation strategy of the table or tables in the dbspace associated with the chunk.

- The root dbspace on the second coserver has a disproportionately high number of disk writes compared to the other dbspaces.

  You might need to define a temporary dbspace on the second coserver if a query is using an implicit temporary table that defaults to the root dbspace because neither the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable is set.

Monitoring Fragmentation on a Specific Coserver

If you suspect that one or two specific coservers are creating performance problems, you can monitor chunks and read and write calls for each fragment on one coserver at a time.

The **xctl -c *n*** prefix to the **onstat** command-line options displays **onstat** information for the coserver that is numbered *n*. You can also run **onstat** commands without **xctl** from a specific coserver node to obtain statistics for only that coserver.

The **sysfragments** system catalog table provides the names of tables that reside in a chunk.

xctl -c n onstat -g iof

The **xctl -c *n* onstat -g iof** option displays the I/O statistics for each chunk or nonchunk file. To display information about the chunks on each coserver, issue this command for each coserver and specify the coserver number after the **-c** flag.

xctl -c n onstat -g ppf

Use the **xctl -c *n* onstat -g ppf** option to display the number of read and write calls for each fragment. To display information about the fragments on each coserver, issue this command on each coserver and specify the coserver number after the **-c** flag.

Although the read and write calls do not show how many disk reads and writes occurred, you can still determine if the I/O activity is balanced across fragments. Compare the number of reads (or writes) for each fragment to see if they are about the same or if some are proportionally greater than others.

The **onstat -g ppf** output does not report the table to which the fragment belongs.

To determine the table name for a table fragment

1. Run **onstat -g ppf** and write down the value that appears in the **partnum** field of its output.
2. Join the **tabid** column in the **sysfragments** system catalog table with the **tabid** column in the **systables** system catalog table to obtain the table name from the **systables** table. Use the **partnum** field value that you obtain in step 1 in the SELECT statement.

```
SELECT a.tabname FROM systables a, sysfragments b
   WHERE a.tabid = b.tabid
      AND partn = partnum_value;
```

The **sysfragments** system catalog table does not store the names of unfragmented tables. Use the following SELECT statement to find the name of an unfragmented table:

```
SELECT tabname FROM SYSTABLES WHERE partnum = partnum_value;
```

sysfragments System Catalog Table

The **sysfragments** system catalog table stores information about fragmented tables. It has a row for each tblspace that holds a table fragment or an index fragment. The **sysfragments** system catalog table includes the following columns that are useful for monitoring the performance of fragmented tables.

Column Name	Description
tabid	Table identifier
indexname	Index identifier
partn	Physical location (tblspace ID)
strategy	Distribution scheme (round-robin, expression based, range based, system derived)
dbspace	Dbspace name for fragment
npused	Number of data pages or leaf pages

(1 of 2)

Column Name	Description
nrows	Number of rows recorded by the most recent UPDATE STATISTICS operation
fragtype	Fragment type: `I` for index, `B` for TEXT or BYTE data, `T` for table data
exprtext	Names of the columns that are hashed If the fragment is created by a hybrid scheme, the hash columns appear first, followed by the fragmentation expression for the dbslice or list of dbspaces
hybdpos	Relative position of the fragment in the set of fragments created by a hybrid fragmentation scheme

(2 of 2)

For a complete list of **sysfragments** columns, see the *Informix Guide to SQL: Reference*.

sysptprof System-Monitoring Interface Table

While a table is open, you can query the **sysptprof** System-Monitoring Interface (SMI) table for current activity information about a tblspace. When the last user who has a table open closes it, all profile statistics are dropped.

You can get the following information directly from **sysptprof**:

- The database name
- The table name
- The partition (tblspace) number
- Lock information
- Read and write information, both page and buffer
- Sequential scan information.

Queries against **sysptprof** can provide useful information about how table fragments are used during queries.

Chapter

10

Queries and the Query Optimizer

In This Chapter

This chapter explains the general principles of query processing. It discusses the following topics:

- The *query plan*, which determines how the database server performs query-execution tasks
- Factors that affect the query plan and how to examine a query to assess these factors
- Operations that take the most time when the database server processes a query
- Optimization of SPL routines

This chapter provides the background information to help you understand the detailed information in the chapters that follow:

- Chapter 11, "Parallel Database Query Guidelines," describes the unique features of parallel database query optimization.
- Chapter 12, "Resource Grant Manager," describes how the RGM manages execution of parallel database queries and explains how you can monitor their use of resources.
- Chapter 13, "Improving Query and Transaction Performance," explains how to improve the performance of specific queries and transactions.

Query Plan

The query optimizer formulates a *query plan* to fetch the data rows that are required to process a query.

The optimizer must evaluate the different ways in which a query might be performed. For example, the optimizer must determine whether indexes should be used. If the query includes a join, the optimizer must determine the join plan (hash or nested loop) and the order in which tables are evaluated or joined. The following section explains the components of a query plan in detail.

Access Plan

The way that the optimizer chooses to read a table is called an *access plan*. The simplest way to access a table is to read it sequentially, which is called a *table scan*. The optimizer chooses a table scan when most of the table must be read or when the table does not have an index that is useful for the query.

The optimizer can also choose to access the table by an index. If the indexed column is the same as a column in a filter of the query, the optimizer can use the index to retrieve only the rows that the query requires. If all of the required columns are in one or more indexes on the table, the optimizer can use a *key-only index scan*. The database server retrieves data from the index but does not access the associated table.

The optimizer compares the costs of each plan to determine the best one. The database server calculates costs from estimates of the number of I/O operations required, calculations required to produce the results, rows accessed, sorting, and so on.

Join Plan

When a query contains more than one table, the tables are usually joined by a WHERE clause, or *filter*, in the query. For example, in the following query, the **customer** and **orders** table are joined by the `customer.customer_num = orders.customer_num` filter:

```
SELECT * from customer, orders
WHERE customer.customer_num = orders.customer_num
AND customer.lname = "SMITH";
```

The way that the optimizer chooses to join the tables is the *join plan*. The join method might be a nested-loop join or a hash join. If a generalized-key (GK) index has been created to contain the result of a join on the tables, the database server uses the GK index instead of joining the table columns again. For more information on the requirements for this kind of index, refer to "Join Indexes" on page 13-26.

Because of the way hash joins work, an application with isolation mode set to Repeatable Read might temporarily lock all the records in tables that are involved in the join, including records that fail to qualify the join. This situation decreases concurrency among connections. Conversely, nested-loop joins lock fewer records but provide inferior performance when a large number of rows is accessed. Thus, each join method has advantages and disadvantages.

Nested-Loop Join

In a nested-loop join, the database server scans the first, or *outer table*, and then joins each of the rows that pass table filters to the rows found in the second, or *inner table*, as shown in Figure 10-2 on page 10-7. To access the outer table, the database server can use an index or scan the table sequentially. The database server applies any table filters first. For each row that satisfies the filters on the table, the database server reads the inner table to find a match.

The database server reads the inner table once for every row in the outer table that fulfills the table filters. Because of the potentially large number of times that the inner table can be read, the database server usually accesses the inner table by an index.

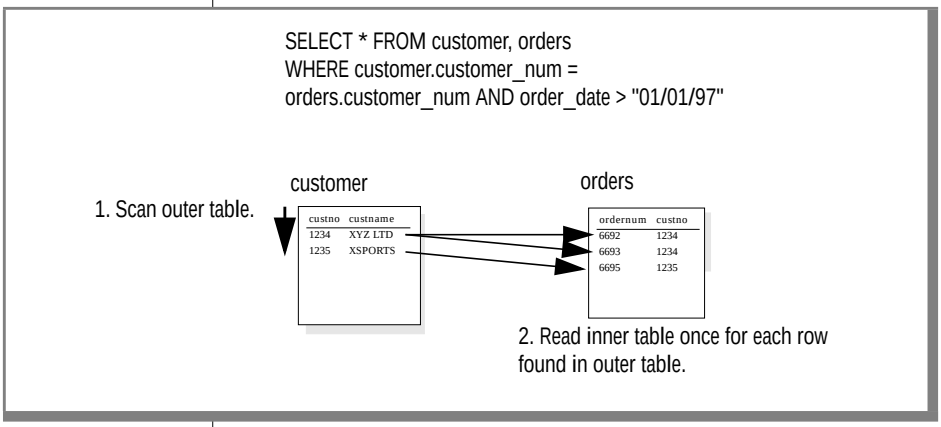


Figure 10-1
Nested-Loop Join

If a table does not have a permanent index, the optimizer can construct an *autoindex* on the table during query execution in order to use the table as an inner table. In some cases, the database server might scan the table, apply any table filters, and put any rows that pass the filters in a temporary table. Then the database server constructs a dynamic index on the temporary table to perform the join.

Hash Join

The optimizer usually uses a hash join when at least one of the two join tables does not have an index on the join column or when the database server must read a large number of rows from either of the tables. No index and no sorting are required when the database server performs a hash join.

A hash join consists of two parts: building the hash table (*build* phase) and probing the hash table (*probe* phase). Figure 10-2 shows a hash join in more detail.

In the build phase, the database server chooses the smaller of the two tables, reads the table and, after applying any filters that would exclude rows from both tables, creates a hash table.

A hash table is like a set of *buckets*. To assign rows to each bucket, the database server applies a hash function on the key value. All rows that evaluate to the same hash value are stored in the same hash bucket.

In the probe phase, the database server reads the other table in the join and applies any filters. For each row that satisfies the filters on the table, the database server applies the same hash function on the join key and probes the hash table in the matching hash bucket to find a matching row in the first table.

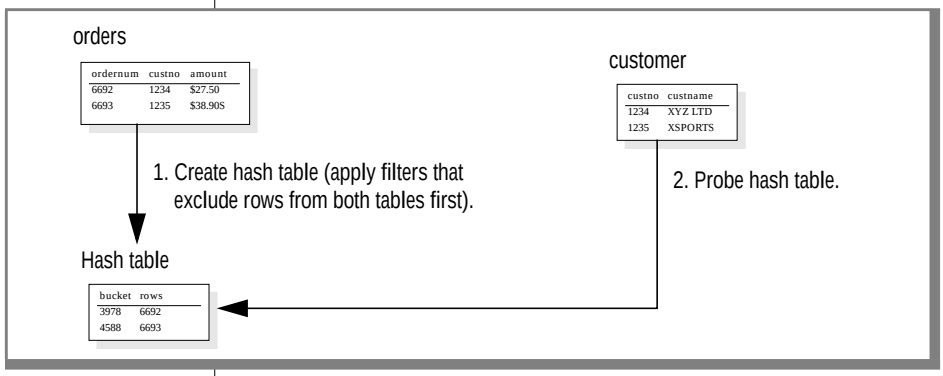


Figure 10-2
How a Hash Join Is Executed

Figure 10-3 shows why a hash join is efficient for joining rows in large tables. The area of the entire square represents the work of processing each row in the **customers** table and the **orders** table, as would be necessary in a nested-loop join.

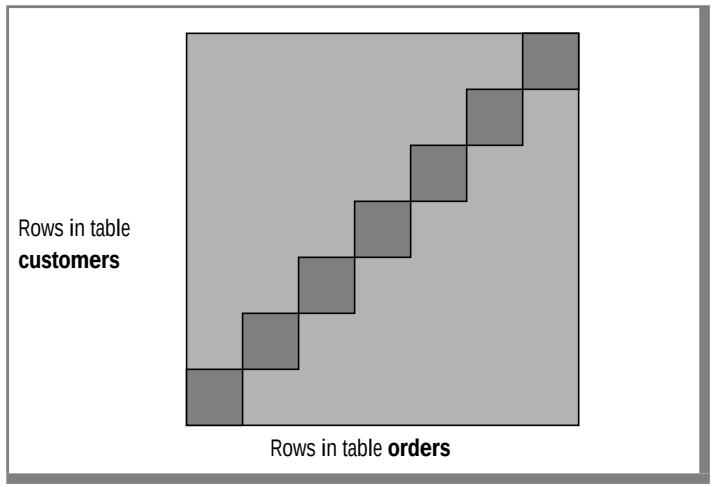


Figure 10-3
Hash-Join Efficiency

The areas of the small squares represent the work required to probe the hash tables created during the build phase by applying a hash function to the join key value in one of the tables.

In the probe phase, the same hash function is applied to the join key in the other table. Then only the hash bucket that contains rows with the same key-value hash result need be probed to match the join key.

The hash buckets, formally called *partitions*, can easily be processed in parallel by several CPU VPs and across several coservers for even greater efficiency.

Smaller hash tables can fit in the virtual portion of the database server shared memory. If the database server runs out of memory, hash tables are stored on disk in the dbspace specified by the DBSPACETEMP configuration parameter or environment variable.

Hash joins and other uses of hash functions are effective in balancing processing or distribution only if the hashed column does not contain many duplicates. For hash joins, if the join key column contains many duplicates, one hash partition will contain many more rows than other partitions. This condition creates one kind of *data skew*. Performance suffers because the other coservers must wait for the coserver that is processing the largest hash table.

Join Order

The order in which the database server joins tables in a query is extremely important. A poor join order can cause poor query performance.

The following section explains how the database server executes a plan according to a specific join order.

Three-Way Join

Consider the following SELECT statement, which calls for a three-way join:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
```

The optimizer can choose one of the following join orders:

- Join **customer** to **orders**. Join the result to **items**.
- Join **orders** to **customer**. Join the result to **items**.
- Join **customer** to **items**. Join the result to **orders**.
- Join **items** to **customer**. Join the result to **orders**.
- Join **orders** to **items**. Join the result to **customer**.
- Join **items** to **orders**. Join the result to **customer**.

Assume that none of the three tables have indexes. Suppose that the optimizer chooses the **customer-orders-items** path and the nested-loop join for both joins. (In reality, the optimizer usually chooses a hash join for two tables without indexes on the join columns.) Figure 10-4 shows the query plan in pseudocode. For information about interpreting query-plan information, see "Display and Interpretation of the Query Plan" on page 10-14.

```
for each row in the customer table do:
   read the row into C
   for each row in the orders table do:
      read the row into O
      if O.customer_num = C.customer_num then
         for each row in the items table do:
            read the row into I
            if I.order_num = O.order_num then
               accept the row and send to user
            end if
         end for
      end if
   end for
end for
```

Figure 10-4
Query Plan with Two Nested-Loop Joins

This procedure reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for each row of the **customer** table
- All rows of the **items** table once for each row of the **customer-orders** pair

This example does not describe the only possible query plan. Another plan merely reverses the roles of **customer** and **orders**: for each row of **orders**, it reads all rows of **customer** to match **customer_num**. It reads the same number of rows in a different order and produces the same set of rows in a different order. In this example, no difference exists in the amount of work that the two possible query plans need to do.

Join with Column Filters

The presence of a *column filter* changes things. A column filter is a WHERE expression that might reduce the number of rows that a table contributes to a join. The following example shows the preceding query with a filter added to reduce the number of rows that the database server must evaluate in the **orders** table:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
      AND O.paid_date IS NULL
```

The expression `O.paid_date IS NULL` filters out some rows, reducing the number of rows that are used from the **orders** table. Consider a plan that starts by reading from **orders**. Figure 10-5 displays it in pseudocode.

```
for each row in the orders table do:
   read the row into O
   if O.paid_date is null then
      for each row in the customer table do:
         read the row into C
         if O.customer_num = C.customer_num then
            for each row in the items table do:
               read the row into I
               if I.order_num = O.order_num then
                  accept row and return to user
               end if
            end for
         end if
      end for
   end if
end for
```

Figure 10-5
Query Plan with One Nested-Loop Join

Let *pdnull* represent the number of rows in **orders** that pass the filter. It is the value of `COUNT(*)` that results from the following query:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

If one customer exists for every order, the plan in Figure 10-5 reads the following rows:

- All rows of the **orders** table once
- All rows of the **customer** table, *pdnull* times
- All rows of the **items** table, *pdnull* times

Figure 10-6 shows an alternative execution plan. It reads from the **customer** table first.

```
for each row in the customer table do:
   read the row into C
   for each row in the orders table do:
      read the row into O
      if O.paid_date is null and
         O.customer_num = C.customer_num then
         for each row in the items table do:
            read the row into I
            if I.order_num = O.order_num then
               accept row and return to user
            end if
         end for
      end if
   end for
```

Figure 10-6
Query Plan with an An Alternative to a Nested Loop

Because the filter is not applied in the first step that Figure 10-6 shows, this plan reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table, once for every row of **customer**
- All rows of the **items** table, *pdnull* times

The query plans in Figure 10-5 and Figure 10-6 produce the same output in a different sequence. They differ in that one reads a table *pdnull* times, and the other reads a table `SELECT COUNT(*) FROM customer` times. By choosing the appropriate plan, the optimizer can save thousands of disk accesses in a real application.

Join with Indexes

The preceding examples do not use indexes or constraints. Indexes and constraints provide the optimizer with options that can greatly improve query-execution speed. Figure 10-7 shows how use of an index might change the query plan for this query:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
      AND O.paid_date IS NULL
```

```
for each row in the customer table do:
   read the row into C
   look up C.customer_num in index on orders.customer_num
   for each matching row in the orders index do:
      read the table row for O
      if O.paid_date is null then
         look up O.order_num in index on items.order_num
         for each matching row in the items index do:
            read the row for I
            construct output row and return to user
         end for
      end if
   end for
end for
```

Figure 10-7
Query Plan with Indexes

Because the keys in an index are sorted, when the database server finds the first matching entry, it can read other rows in order without further searching because they are located in physically adjacent positions. This query plan reads only the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once (because each order is associated with only one customer)
- Only rows in the **items** table that match *pdnull* rows from the **customer-orders** pairs

This query plan achieves a large reduction in effort compared with plans that do not use indexes. An inverse plan, which reads **orders** first and looks up rows in the **customer** table by **customer** table index, is also feasible by the same reasoning.

Using an index incurs one additional cost over reading the table sequentially. The database server must locate each entry or set of entries with the same value in the index. Then, for each entry in the index, the database server must access the table to read the associated row.

The physical order of rows in a table also affects the cost of index use. To the degree that a table is ordered relative to an index, the overhead of accessing multiple table rows in index order is reduced. For example, if the **orders** table rows are physically ordered according to the customer number, the database server retrieves multiple orders for a given customer faster than if the table rows are ordered randomly.

Display and Interpretation of the Query Plan

To display the query plan, any user who runs a query can execute the SET EXPLAIN ON statement before running the query. After the database server executes the SET EXPLAIN ON statement, it writes an explanation of each query plan to a file called **sqexplain.out**. The **sqexplain.out** file includes information about how the database server used threads to execute a parallel query and what SQL operators it used. For more information, see "SQL Operators" on page 11-6.

Examining query plans can be useful for analysis of OLTP transactions as well as complex DSS queries.

For example, the following partial **sqexplain.out** output shows that the database server executed a specific query with two hash joins. The report lists the tables in their join order.

```
QUERY:
------
SELECT i.stock_num FROM items i, stock s, manufact m
        WHERE i.stock_num = s.stock_num
        AND i.manu_code = s.manu_code
        AND s.manu_code = m.manu_code

Estimated Cost: 1
Estimated # of Rows Returned: 9

1) informix.m: SEQUENTIAL SCAN

2) informix.s: SEQUENTIAL SCAN


DYNAMIC HASH JOIN (Build Outer Broadcast)
    Dynamic Hash Filters: informix.m.manu_code = informix.s.manu_code

3) informix.i: SEQUENTIAL SCAN


DYNAMIC HASH JOIN (Build Inner Broadcast)
    Dynamic Hash Filters: (informix.s.stock_num = informix.i.stock_num AND
informix.s.manu_code = informix.i.manu_code )
```

The Build Outer clause in this example means that the result of the hash join of tables **m** and **s** is used as the build table. If this hash join is the first, the Build Inner clause means that the resulting table is created with a base table as input so that the Build Inner step can be created in memory before the Build Outer step is complete. The database server broadcasts both tables to each join thread to prevent data skew.

The following partial **sqexplain.out** output shows that the database server read the **customer** table using the index on **customer_num** and created a temporary table to order the query results:

```
QUERY:
------
SELECT fname,lname,company FROM customer
        WHERE company MATCHES 'Sport*' AND customer_num
        BETWEEN 110 AND 115
        ORDER BY lname

Estimated Cost: 1
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

1) informix.customer: INDEX PATH

    Filters: informix.customer.company MATCHES 'Sport*'

    (1) Index Keys: customer_num   (Parallel, fragments: ALL)
        Lower Index Filter: informix.customer.customer_num >= 110
        Upper Index Filter: informix.customer.customer_num <= 115
```

Query Plans for Subqueries

A SELECT statement nested in the WHERE clause of another SELECT statement, or in an INSERT, DELETE, or UPDATE statement, is called a subquery.

Subqueries can be *correlated* or *uncorrelated*. A subquery, which is an inner SELECT statement, is correlated when the value that it produces depends on a value produced by the outer SELECT statement that contains it. Any other kind of subquery is considered uncorrelated.

Because the result of the subquery might be different for each row that the database server examines, the subquery begins anew for every row in the outer query if the current correlation values are different from the previous ones. The optimizer tries to use an index on the correlation values to cluster identical values together. This process can be extremely time consuming.

The optimizer can often rewrite a correlated subquery to unnest it in one of the following ways:

- As a join query

  A join executes as fast as or faster than a correlated subquery. The optimizer might use other SQL operators, such as GROUP and INSERT, to ensure correct semantics of the query. These additional SQL operators can execute in parallel to improve the response time of the correlated subquery. For more information about SQL operators, refer to "SQL Operators" on page 11-6.

- As separate queries

  Two separate queries can execute faster than a correlated subquery because each query is optimized separately. The optimizer uses fragmented temporary tables to hold the interim results.

The optimizer can unnest most correlated subqueries if the rewritten query provides a lower cost. Even if the optimizer cannot unnest a correlated subquery, the database server can speed execution of the query with parallel execution.

For information on how correlated subqueries execute in parallel, refer to "Parallel Execution of Nested Correlated Subqueries" on page 11-25. If possible, the application designer should avoid writing nested subqueries.

If the optimizer decides to rewrite the correlated subquery, the output of the SET EXPLAIN ON statement shows how the subquery is rewritten. For example, the partial **sqexplain.out** output in Figure 10-8 shows a subquery that the optimizer rewrites as the outer table in a hash join and *broadcasts* a copy of the table to every join thread to prevent data skew. This **sqexplain.out** output shows that the optimizer uses the GROUP operator to ensure correct semantics of the query.

Figure 10-8
Rewritten Subquery as a Join in sqexplain.out Output

```
QUERY:
------
select a from tab1
where a in (select a from tab2 where tab2.b = 50)

Estimated Cost: 3
Estimated # of Rows Returned: 1

1) virginia.tab2: SEQUENTIAL SCAN

    Filters: virginia.tab2.b = 50

2) virginia.tab1: SEQUENTIAL SCAN


DYNAMIC HASH JOIN (Build Outer Broadcast)
    Dynamic Hash Filters: virginia.tab2.a = virginia.tab1.a

# of Secondary Threads = 4

XMP Query Plan

  oper       segid  brid   width
  ----------------------------
  scan       3      0      1
  scan       4      0      1
  hjoin      2      0      1
  group      2      0      1
  group      1      0      1
```

Figure 10-9 shows a query that the optimizer rewrites into a series of queries with joins.

Figure 10-9
Rewritten Query as Series of Queries with Joins in sqexplain.out Output

```
QUERY:
------
select * from tab1
where tab1.a IN (
select tab2.a from tab2 where tab1.b = tab2.b
and tab2.c IN (
select tab3.c from tab3 where tab3.a = tab1.a
and tab3.b = tab2.b ))
Estimated Cost: 14

Estimated # of Rows Returned: 10
1) virginia.tab1: SEQUENTIAL SCAN
2) virginia.tab3: SEQUENTIAL SCAN

DYNAMIC HASH JOIN (Build Outer Broadcast)
   Dynamic Hash Filters: (virginia.tab1.b = virginia.tab3.b AND virginia.tab1.a =
virginia.tab3.a )
3) virginia.tab2: SEQUENTIAL SCAN

DYNAMIC HASH JOIN (Build Inner Broadcast)
   Dynamic Hash Filters: virginia.tab1.b = virginia.tab2.b
# of Secondary Threads = 7
XMP Query Plan
  oper       segid  brid   width
  -----------------------------
  scan       5      0      1
  scan       6      0      1
  hjoin      4      0      1
  scan       7      0      1
  hjoin      3      0      1
  group      3      0      1
  group      2      0      1
  flxins     1      0      1

QUERY:
------
(continued...)
Estimated Cost: 14
Estimated # of Rows Returned: 10
1)  subqtmp.0x60478018: SEQUENTIAL SCAN
# of Secondary Threads = 2

XMP Query Plan
 oper       segid  brid   width
  -----------------------------
  scan       2      0      1
  group      2      0      1
  group      1      0      1
```

The optimizer uses the FLEX INSERT SQL operator to stage temporary results. The FLEX INSERT SQL operator can execute parallel inserts into the fragmented temporary table. For more information, refer to "Fragmenting Temporary Tables" on page 9-58. For more information on how the database server uses SQL operators for parallel execution, refer to "Structure of Query Execution" on page 11-5.

Query-Plan Evaluation

When the optimizer determines the query plan, it assigns a cost to each possible plan and then chooses the lowest-cost plan. Some of the factors that the optimizer uses to create a query plan include:

- the number of I/O requests that are associated with each file-system access.
- the CPU work that is required to determine which rows meet the query predicate.
- the resources that are required to sort or group the data.
- the table statistics that are stored in the system catalog.

The optimizer considers all query plans by analyzing factors such as disk I/O and CPU costs. It constructs all feasible plans simultaneously using a bottom-up, breadth-first search strategy. That is, the optimizer first constructs all possible join pairs. It eliminates the more expensive of any *redundant* pair, which is a join pair that contains the same tables and produces the same set of rows as another join pair. For example, if neither join specifies an ordered set of rows by using the ORDER BY or GROUP BY clauses of the SELECT statement, the join pair (A x B) is equivalent to (B x A). Because either join produces the same rows, the optimizer does not need to determine which is more expensive.

If the query uses additional tables, the optimizer joins each remaining pair to a new table to form all possible join triplets, eliminating the more expensive of redundant triplets, and so on for each additional table to be joined. When a nonredundant set of possible join combinations has been generated, the optimizer selects the plan that appears to have the lowest execution cost.

Statistics Used to Calculate Costs

The optimizer uses system catalog statistics and information about the available indexes to calculate costs for each query plan. The statistics describe the characteristics of the table data and indexes, including the number of rows in a table and how the column values are distributed.

The accuracy with which the optimizer can assess the execution cost of a query plan depends on how accurate the information is and how recently it was gathered. Use the UPDATE STATISTICS statement to maintain statistical information about a table and its associated indexes. Updated statistics provide the query optimizer with information that can minimize the amount of time required to perform queries on that table.

The database server creates a statistical profile of a table when the table is created. This profile is refreshed only when you issue the UPDATE STATISTICS statement. To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals. The query optimizer does not recalculate the profile for tables.

In some cases, updating the statistics might take longer than executing a query. For information about improving UPDATE STATISTICS performance, refer to "Using Indexes" on page 13-13.

The optimizer uses the following system catalog information as it creates a query plan:

- The number of rows in a table, as calculated the last time that UPDATE STATISTICS was run
- Unique column constraints
- The distribution of column values, when requested with the MEDIUM or HIGH keyword in the UPDATE STATISTICS statement

  For more information on data distributions, refer to "Creating Data-Distribution Statistics" on page 13-9.
- The number of disk pages that contain row data

In addition, the optimizer uses available table statistics and index type to determine costs associated with index accesses.

For more information on system catalog tables, refer to the *Informix Guide to SQL: Reference*.

Query Evaluation

Before the optimizer estimates the cost of each possible query plan, it must examine the query filters and the indexes that could be used in the plan.

Filter Selectivity Evaluation

The optimizer bases query-cost estimates on the number of rows to be retrieved from each table. In turn, the estimated number of rows is based on the *selectivity* of each conditional expression in the WHERE clause. A conditional expression that is used to select particular rows is called a *filter*. The selectivity is a value between `0` and `1` that indicates the proportion of rows in the table that the filter can pass. A very selective filter, one that passes few rows, has a selectivity near `0`; a filter that passes almost all rows has a selectivity near `1`. For guidelines on filters, see "Improving Filter Selectivity" on page 13-27.

The following table lists some common selectivity filters that the optimizer assigns to filters of different types. The list is not exhaustive. Selectivities calculated using data distributions are more accurate than those shown in the table. However, if data distribution information from UPDATE STATISTICS is not available, the optimizer calculates selectivities for the following filters based on table indexes.

Filter Expression	Selectivity (F)
indexed-col = *literal-value* *indexed-col* = *host-variable* *indexed-col* IS NULL	F = 1 / (number of distinct keys in index)
tab1.indexed-col = *tab2.indexed-col*	F = 1 / (number of distinct keys in the larger index)
indexed-col > *literal-value*	F = (*2nd-max* - *literal-value*) / (*2nd-max* - *2nd-min*)
indexed-col < *literal-value*	F = (*literal-value* - *2nd-min*) / (*2nd-max* - *2nd-min*)
any-col IS NULL *any-col* = *any-expression*	F = 1 / 10

(1 of 2)

Filter Expression	Selectivity (F)
any-col > *any-expression* *any-col* < *any-expression*	F = 1 / 3
any-col MATCHES *any-expression* *any-col* LIKE *any-expression*	F = 1 / 5
EXISTS *subquery*	F = 1 if *subquery* estimated to return >0 rows, else 0
NOT *expression*	F = 1 - F(*expression*)
expr1 AND *expr2*	F = F(*expr1*) × F(*expr2*)
expr1 OR *expr2*	F = F(*expr1*) + F(*expr2*) - (F(*expr1*) × F(*expr2*))
any-col IN *list*	Treated as *any-col* = $item_1$ OR…OR *any-col* = $item_n$
any-col relop ANY *subquery*	Treated as *any-col relop* $value_1$ OR…OR *any-col relop* $value_n$ for estimated size of subquery *n*

Key:

- *indexed-col*: first or only column in an index
- *2nd-max*, *2nd-min*: second-largest and second-smallest key values in indexed column
- *any-col*: any column not covered by a preceding formula

(2 of 2)

Index Evaluation

The optimizer notes whether an indexed column can be used to evaluate a filter. For this purpose, an indexed column is any single column with an index or the first column named in a composite index.

The optimizer can use an index in the following cases:

- When the column is indexed and a value to be compared is a literal, a host variable, or an uncorrelated subquery

  To locate relevant rows in the table, the database server first looks for the row in an appropriate index. If an appropriate index is not available and the database server cannot eliminate table fragments based on the fragmentation scheme, the database server must completely scan each table.

- When the column is indexed and the value to be compared is a column in another table (a join expression)

  The database server can use the index to find matching values. The following join expression shows such an example:

  ```
  WHERE customer.customer_num = orders.customer_num
  ```

  If rows of **customer** are read first, values of **customer_num** can be applied to an index on **orders.customer_num**.

- When processing an ORDER BY clause

  If all the columns in the clause appear in the required sequence in a single index, the database server can use the index to read the rows in their ordered sequence, thus avoiding a sort.

- When processing a GROUP BY clause

  If all the columns in the clause appear in one index, the database server can read groups with equal keys from the index without requiring additional processing after the rows are retrieved from their tables.

- When queries frequently access only a subset of rows

  For more information, refer to "Selective Indexes" on page 13-24.

- When an expression is frequently used in a filter

  For more information, refer to "Virtual-Column Index" on page 13-25.

- When a specific join of tables is frequently used in queries

  For more information, refer to "Join Indexes" on page 13-26.

For detailed information about how the optimizer might choose to use indexes, see "Using Indexes" on page 13-13.

Time Costs of a Query

This section explains the response-time effects of actions that the database server performs as it processes a query.

You cannot adjust the construction of the query to reduce some of the costs described in this section. However, you can reduce the following costs by optimal query construction, appropriate table fragmentation, and appropriate indexes:

- Sort time
- Data mismatches
- In-place ALTER TABLE
- Index lookups

For information about how to optimize specific queries, see Chapter 13, "Improving Query and Transaction Performance."

Memory-Activity Costs

The database server can process only data that is in memory. It must read rows into memory to evaluate those rows against the filters of a query. When rows that satisfy those filters are found, the database server prepares an output row in memory by assembling the selected columns.

The database server performs most of these tasks very quickly. Depending on the computer and its workload, the database server can perform hundreds or even thousands of comparisons each second. As a result, the time spent on in-memory work is usually a small part of the execution time.

Although some in-memory activities, such as sorting, take a significant amount of time, it takes much longer to read a row from disk than to examine a row that is already in memory.

Sort-Time Costs

A sort requires in-memory work as well as disk work. The in-memory work depends on the number of columns that are sorted, the width of the combined sort key, and the number of row combinations that pass the query filter. You can use the following formula to calculate the in-memory work that a sort operation requires:

$$W_m = (c * N_{fr}) + (w * N_{fr} \log_2(N_{fr}))$$

W_m is the in-memory work.

c is the number of columns to order. It represents the costs to extract column values from the row and concatenate them into a sort key.

w is proportional to the width of the combined sort key in bytes. It stands for the work to copy or compare one sort key. The numeric value for *w* depends strongly on the computer hardware in use.

N_{fr} is the number of rows that pass the query filter.

Sorting might require writing information temporarily to disk if there is a large amount of data to sort. You can direct the disk writes to the operating-system file space or a dbspace that the database server manages. For details, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

The disk work depends on the number of disk pages where rows appear, the number of rows that meet the conditions of the query predicate, the number of rows that can be placed on a sorted page, and the number of merge operations that must be performed. Use the following formula to calculate the disk work that a sort operation requires:

$$W_d = p + (N_{fr}/N_{rp}) * 2 * (m-1))$$

W_d is the disk work.

p is the number of disk pages.

N_{fr} is the number of rows that pass the filters.

N_{rp} is the number of rows that can be placed onto a page.

m represents the number of *levels of merge* that the sort must use.

The factor *m* depends on the number of sort keys that memory can contain. If there are no filters, N_{fr}/N_{rp} is equivalent to *p*.

When memory can contain all the keys, *m*=1 and the disk work is equivalent to *p*. In other words, the rows are read and sorted in memory.

For moderate-sized to large tables, rows are sorted in batches that fit in memory, and then the batches are merged. When *m*=2, the rows are read, sorted, and written in batches. Then the batches are read again and merged, resulting in disk work proportional to the following value:

$W_d = p + (2 * (N_{fr}/N_{rp}))$

The more specific the filters, the fewer the rows that are sorted. As the number of rows increases, and the amount of memory decreases, the amount of disk work increases.

To reduce the cost of sorting, use the following methods:

- Make your filters as specific (selective) as possible.
- Limit the projection list to the columns that are relevant to your problem.

Row-Reading Costs

When the database server needs to examine a row that is not already in memory, it must read that row from disk. The database server does not read only one row; it reads the entire page that contains the row. If the row spans more than one page, it reads all of the spanned pages.

The actual cost of reading a page is variable and hard to predict. It is a combination of the following factors.

Factor	Implications
Buffering	If the needed page is in a page buffer already, the cost of access is nearly 0.
Contention	If two or more applications require access to the disk hardware, I/O requests can be delayed.
Seek time	The slowest part of disk access is *seek time*, which is moving the access arm to the track that holds the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts. Seek time varies from 0 to nearly a second.
Latency	The transfer cannot start until the beginning of the page rotates under the access arm. This rotational delay depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from 0 to a few milliseconds.

The time cost of reading a page can vary from microseconds for a page that is already in a buffer, to a few milliseconds when contention is 0 and the disk arm is already in position, to hundreds of milliseconds when the page is in contention and the disk arm is over a distant cylinder of the disk.

Sequential-Access Costs

Disk costs are lowest when the database server reads the rows of a table in physical order. When the first row on a page is requested, the disk page is read into a buffer page. Once the page is read in, it need not be read again; requests for subsequent rows on that page are filled from the buffer until all the rows on that page are processed. After one page has been read, the page for the next set of rows must be read. To make sure that the next page is ready in memory, use the read-ahead configuration parameters described in "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-28.

When you use raw devices for dbspaces and the table is organized properly, the disk pages of consecutive rows are placed in consecutive locations on the disk. With this arrangement, the access arm moves little to read rows sequentially. In addition, latency costs are usually lower when pages are read sequentially.

Nonsequential-Access Costs

Whenever a table is read in random order, additional disk accesses are required to read the rows in the required order. Disk costs are higher when the rows of a table are read in a sequence unrelated to physical order on disk. Because the pages are not read sequentially from the disk, both seek and rotational delays occur before each page can be read. As a result, the disk-access time is much higher when table rows are read nonsequentially than when the same table is read sequentially.

Nonsequential access often occurs when you use an index to locate rows. Although index entries are sequential, there is no guarantee that rows with adjacent index entries must reside on the same (or adjacent) data pages. In many cases, a separate disk access must be made to fetch the page for each row located through an index. If a table is larger than the page buffers, a page that contained a row previously read might be cleaned (removed from the buffer and written back to the disk) before a subsequent request for another row on that page can be processed. That page might have to be read in again.

Depending on the relative ordering of the table with respect to the index, you can sometimes retrieve pages that contain several needed rows. The degree to which the physical ordering of rows on disk corresponds to the order of entries in the index is called *clustering*. A highly clustered table is one in which the physical ordering on disk corresponds closely to the index.

Index-Lookup Costs

The database server incurs additional costs when it finds a row through an index. The index is stored on disk, and its pages must be read into memory along with the data pages that contain the desired rows.

An index lookup works down from the root page to a leaf page. (See "Managing Indexes" on page 7-11.) The root page, because it is used so often, is almost always found in a page buffer. The odds of finding a leaf page in a buffer depend on the size of the index, the form of the query, and the frequency of column-value duplication. If each value occurs only once in the index and the query is a join, each row to be joined requires a nonsequential lookup into the index, followed by a nonsequential access to the associated row in the table.

If many duplicate rows exist for each distinct index value and the associated table is highly clustered, the added cost of joining through the index can be slight.

In-Place ALTER TABLE Costs

When you execute an ALTER TABLE statement, the database server might use an in-place alter algorithm to modify each row when it is inserted instead of using a slow or fast ALTER TABLE operation. After you execute the ALTER TABLE statement, the database server uses the new definition when it inserts rows.

If your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query because the database server reformats each row before it is returned.

For more information on the conditions and performance advantages when an in-place ALTER TABLE occurs, refer to "Altering a Table Definition" on page 6-34.

View Costs

You can create views of tables for a number of reasons:

- To limit the data that a user can access
- To reduce the time that it takes to write a complex query
- To hide the complexity of the query that a user needs to write

However, a query against a view might execute more slowly than expected if the complexity of the view definition requires a temporary table to be created to process the query. For example, a query against a a view that involves a union to combine results from several SELECT statements causes the database server to create a temporary table.

The following sample SQL statement creates a view that includes unions:

```
CREATE VIEW view1 (col1, col2, col3, col4)
   AS
      SELECT a, b, c, d
         FROM tab1 WHERE ...
      UNION
      SELECT a2, b2, c2, d2
         FROM tab2 WHERE ...
      ...
      UNION
      SELECT an, bn, cn, dn
         FROM tabn WHERE ...
;
```

When you create a view that contains complex SELECT statements, the end user does not need to manage the complexity. The user can write a simple query, as the following example shows:

```
SELECT a, b, c, d
   FROM view1
      WHERE a < 10;
```

However, this query against **view1** might execute more slowly than the simplicity of the query might imply because the database server creates a fragmented temporary table for the view before it executes the query.

To find out if you have a query that must build a temporary table to materialize the view, execute the SET EXPLAIN statement in SQL and look at the **sqexplain.out** file. If `Temp Table For View` appears in the **sqexplain.out** file, your query requires a temporary table to materialize the view.

Small-Table Costs

A table is small if it occupies so few pages that it can be retained entirely in the page buffers. Operations on small tables are generally faster than operations on large tables.

For example, a **state** table that relates postal-code abbreviations to names of states might have a total size less than 1,000 bytes and fit in no more than two pages. You can include this table in any query at little cost. No matter how the query uses this table, it costs no more than two disk accesses to retrieve the whole table from disk the first time that it is required.

To make sure that a small, often-used table remains in memory as long as possible, specify the SQL statement SET Residency for the table, as described in "Keeping Small Tables in Memory" on page 6-47.

Data-Mismatch Costs

An SQL statement can encounter additional costs when the data type of a column that is used in a condition differs from the definition of the column in the CREATE TABLE statement.

For example, the following query contains a condition that compares a column to a data type value that differs from the table definition:

```
CREATE TABLE table1 (a integer, ...);
SELECT * FROM table1
   WHERE a = '123';
```

The database server rewrites this query before execution to convert `'123'` to an integer. The **sqexplain.out** output shows the query in its adjusted format. This data conversion requires no noticeable overhead.

The additional costs of a data mismatch are highest when the query compares a character column with a noncharacter value and the length of the number is not equal to the length of the character column. For example, the following query contains a condition in the WHERE clause that equates a character column to an integer value because of missing quotation marks:

```
CREATE TABLE table2 (char_col char(3), ...);
SELECT * FROM table2
   WHERE char_col = 1;
```

This query finds all of the following values that are valid for **char_col**:

```
' 1'
'001'
'1'
```

These values are not necessarily clustered together in the index keys. Therefore, the index does not provide a fast and correct way to obtain the data. The **sqexplain.out** file shows a sequential scan for this situation.

Warning: *The database server does not use an index when the SQL statement compares a character column with a noncharacter value that is not equal in length to the character column.*

GLS

GLS Functionality Costs

Sorting and indexing certain data sets can cause significant performance degradation. If you do not need a non-ASCII collation sequence, Informix recommends that you use the CHAR and VARCHAR data types for character columns whenever possible. Because CHAR and VARCHAR data require simple value-based comparison, sorting and indexing these columns is less expensive than for non-ASCII data types (NCHAR or NVARCHAR, for example). For more information on other character data types, see the *Informix Guide to GLS Functionality*.

Fragmentation Costs

How the data is fragmented across coservers and dbspaces can affect performance, especially in large decision support queries. At a certain point, the overhead for coordinating scan operations across coservers might offset the gains from fragmentation, particularly for small fragments, unless you fragment tables to take advantage of either of the following features:

- Collocated joins

  When you join two tables spread across coservers, you can achieve better performance by using collocated joins. For more information about collocated joins, refer to "Ensuring Collocated Joins" on page 9-30.

- Fragment elimination

  To reduce response time for a query, you can eliminate fragments from a query search. For more information, refer to "Designing Distribution for Fragment Elimination" on page 9-41.

SQL in SPL Routines

SPL routines can improve performance of OLTP transactions because SQL statements in SPL routines are optimized once and cached in memory for repeated use during a single session.

The following section contains information about how and when the database server optimizes and executes SQL in an SPL routine.

Optimization of SQL

If an SPL routine contains SQL statements, the query optimizer evaluates the possible query plans for SQL in the SPL routine and selects the query plan with the lowest cost. The database server puts the selected query plan for each SQL statement in an execution plan for the SPL routine.

To avoid unnecessary reoptimization, query plans in an SPL routine are cached for each session. If the query plan is not in the session cache, the database server optimizes each SQL statement immediately before it executes the statement in the SPL routine.

Execution of SPL Routines

When the database server executes an SPL routine with the EXECUTE PROCEDURE statement, with the SPL CALL statement, or in an SQL statement, the following activities occur:

- The database server reads the interpreter code from the system catalog tables and converts it from a compressed format to an executable format.
- The database server executes any SPL statements that it encounters.
- When the database server encounters an SQL statement, it parses, optimizes, and executes the statement. If the query plan for the statement has been cached for the session, the statement is not reoptimized.
- When the database server reaches the end of the SPL routine or when it encounters a RETURN statement, it returns any results to the client application. Unless the RETURN statement has a WITH RESUME clause, the SPL routine execution is complete.

The database server executes the SPL routine once for each EXECUTE PROCEDURE statement. However, the database server can execute a procedure many times if it is in a SELECT statement. The following example calls **get_order_total** once for each row found in the **customer** table:

```
SELECT customer_num, get_order_total(customer_num)
FROM customer;
```

If the SPL routine is in the WHERE clause of the SELECT statement, and it contains no parameters or uses only constants as parameters, the database server executes the SPL routine only once. The following example executes **get_first_day** only once:

```
SELECT order_num, order_date
FROM orders
WHERE order_date > get_first_day();
```

Not only is a SPL routine of this kind executed only once, but it is *pre-executed*. The database server executes the procedure and replaces the returned value in the SQL statement.

```
SELECT order_num, order_date
FROM orders
WHERE order_date > "10/01/97"
```

Chapter

11

Parallel Database Query Guidelines

In This Chapter

Extended Parallel Server is designed to execute queries with *parallel processing* whenever appropriate. When the database server processes a query in parallel, it divides the work into subtasks and processes the subtasks simultaneously on several processors and coservers. It processes memory-intensive queries in parallel. For example, if a query requires joining tables or sorting data, the optimizer can process the work in parallel on several processors and on several coservers. Queries processed in parallel are sometimes called *PDQ queries*.

Information in this chapter should help you interpret the output of the various **onstat** utility commands and the output that the optimizer produces when you use the SQL statement SET EXPLAIN ON. For information about **onstat** utility commands, see "Monitoring Query Resource Use" on page 12-18.

This chapter covers the following topics:

- Parallel processing concepts
- When the database server uses parallel processing
- When the database server does not use parallel processing

Table fragmentation, which allows you to separate a table into fragments that are stored on different coservers and disks, enhances the benefits of parallel processing. parallel processing delivers maximum performance benefits when the data that the query requires is in fragmented tables. For information about how to combine the advantages of parallel processing and fragmentation, see "Planning a Fragmentation Strategy" on page 9-6.

Whenever a query requires information from table fragments on more than one coserver or requires resources that exceed the capacity of the local coserver, the database server uses its parallel processing capabilities. For information about when queries are not processed in parallel, see "SQL Operations That Are Not Processed in Parallel" on page 11-27.

Parallel Database Queries

Parallel processing refers to the techniques that the optimizer uses to distribute the execution of a single query over several processors and coservers. The optimizer uses parallel processing for queries that require large amounts of resources, in particular large quantities of memory.

When the database server processes a query in parallel, it first divides the query into subplans that might be processed on several coservers and CPUs. The database server then allocates the subplans to threads that process the subplans in parallel. Because each subplan represents a smaller amount of processing time when compared to the entire query, and because each subplan is processed simultaneously with all other subplans, queries are processed faster. Figure 11-1 illustrates how subplans might be processed.

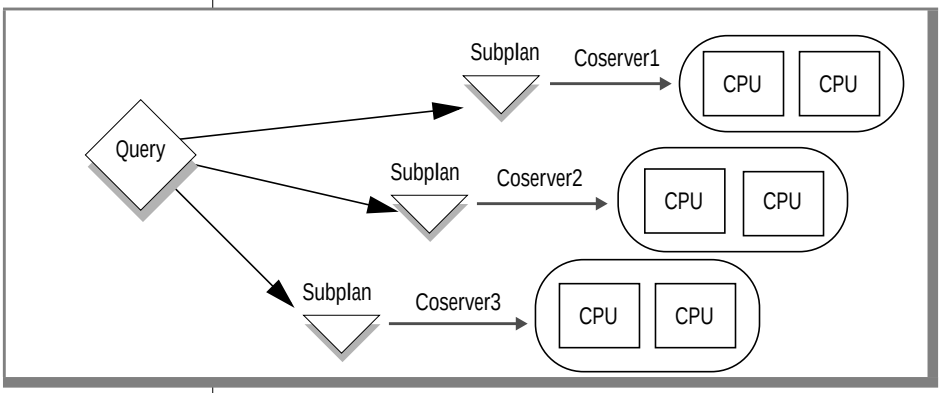


Figure 11-1
Parallel Database Query (PDQ)

High Degree of Parallelism

The degree of parallelism for a query refers to the number of instances that the optimizer executes in parallel to run the query. For example, a two-table join executed by six instances, with each instance executing one-sixth of the required processing, has a higher degree of parallelism than one executed by two instances.

The database server determines the best parallel-processing plan for a query. It bases its plan on the fragmentation of the tables that are being queried, whether it can ignore some fragments, and the complexity of the query, as well as the number of available coservers, the number of virtual processors (VPs) on each coserver, and other resources.

Except for transactions that are not memory intensive and retrieve only a few specified rows from a table fragment, SQL operations are completely parallel. *Completely parallel* means that query execution occurs in multiple instances simultaneously on all CPU VPs across all coservers.

The database server also processes the query components themselves in parallel. For example, consider the way the database server uses pipes to process a complex join. First, the database server scans the data in parallel. As soon as it has scanned enough data, it begins the join. Immediately after the join begins, the database server begins the sort and other required processing, which continues until the full join is complete.

The greatest advantage of parallel processing occurs when you fragment tables across coservers that are multiprocessor computers or other configurations that allow extensive parallel processing. However, performance gains occur even with nonfragmented tables on a uniprocessor computer.

Structure of Query Execution

The optimizer divides a query into components that it can perform in parallel by VPs across coservers to increase the speed of query execution significantly.

Depending on the number of tables or fragments that a query must search, the optimizer determines if a query subplan can execute in parallel. If the words `Parallel, fragments` appear in the access plan for a table in the SET EXPLAIN output, parallel scans occur.

The Resource Grant Manager (RGM) assigns the query operators to different instances across coservers. The SET EXPLAIN output lists these instances as *secondary threads*. For more information on the RGM, refer to Chapter 12, "Resource Grant Manager."

Secondary threads are classified as either *producers* or *consumers*, depending on their function. A producer thread supplies data to another thread. For example, a scan thread might read data from shared memory that corresponds to a given table and pass it to a join thread. In this case, the scan thread is considered a producer, and the join thread is considered a consumer. The join thread, in turn, might pass data to a sort thread. In that case, the join thread is considered a producer, and the sort thread is considered a consumer.

The optimizer uses *SQL operators* and *exchange operators* to execute a query, as the following sections explain.

SQL Operators

An *SQL operator* is a process that performs a small number of predefined operations on one or more streams of rows.

Figure 11-2 shows the different types of SQL operators and how they might be used in parallel processing. Use this list of operators to interpret the SET EXPLAIN ON and **onstat** utility output.

Figure 11-2
Function and Parallel Execution of SQL Operators

SQL Operator	Function	Parallel Instances
ANTIJOIN	Removes duplicate and incorrect rows that might be produced by a nested loop join using a outer join or occasionally by a hash join when the rows are joined on different instances.	Yes
EXPRESSION	Evaluates an expression that is a subquery or calls an SPL routine to determine the best query plan.	No
FLEX INSERT	Inserts rows into an automatically created temporary table for a SELECT... INTO TEMP statement.	Yes
FLEX JOIN	Used only to balance data skew in the query processing. The optimizer creates parallel instances of the FLEX JOIN operator on available coservers. Flex joins must be enabled explicitly for a session. For information about how to enable flex joins, refer to your release notes.	Yes
GROUP	Groups the data for GROUP BY clause, aggregates, and DISTINCT processing.	Yes
HASH JOIN	Uses a hash method to join two tables. Uses one table to build a hash table and joins the other data table to it. For an explanation of the advantages of hash joins, see "Hash Join" on page 10-6.	Yes
INSERT	Inserts rows into a local table or index fragment or into an unfragmented table or index for a INSERT statement.	Yes
NESTED LOOP JOIN	Performs the standard nested-loop join logic.	Yes

(1 of 2)

SQL Operator	Function	Parallel Instances
PROJECT	Obtains values in the projection list of the SELECT statement.	Usually
SCAN	Reads a local table or index fragment or an unfragmented table or index sequentially.	Yes
SORT	Orders the data.	Yes

(2 of 2)

The optimizer uses SQL operators and exchanges to divide a query plan and construct a tree of operators. For information about e*xchanges*, see "Exchanges" on page 11-10. This operator tree consists of a branch and a branch instance:

- Branch (sometimes referred to as *segment*)

  Each branch represents a set of one or more SQL operators that do not have exchanges between them. For example, a branch might contain a group operator and a hash-join operator.

- Branch instance

  A branch instance is a secondary thread that is actually executing one of the SQL operators in the branch.

  The optimizer creates multiple instances of each branch to execute in parallel on different parts of the data.

  The SCAN and INSERT operators execute in parallel, based on the fragmentation strategy of the tables and indexes. For information about the other SQL operators that execute in parallel, such as FLEX INSERT, FLEX JOIN, and GROUP, see Figure 11-2 on page 11-7. The number of instances of these SQL operators is determined by the availability and number of CPU VPs.

Some SQL operators handle data from a *local table* or *local index*. A table or index is local if it resides on the same coserver where the SQL operator is executing. The INSERT and SCAN SQL operators handle local data.

The optimizer structures complex queries into a plan of SQL operators. Figure 11-3 on page 11-9 shows the SQL operator plan that the optimizer constructs to process the following SQL query:

```
SELECT geo_id, sum(dollars)
   FROM customer a, cash b
   WHERE a.cust_id=b.cust_id
   GROUP BY geo_id
   ORDER BY SUM(dollars);
```

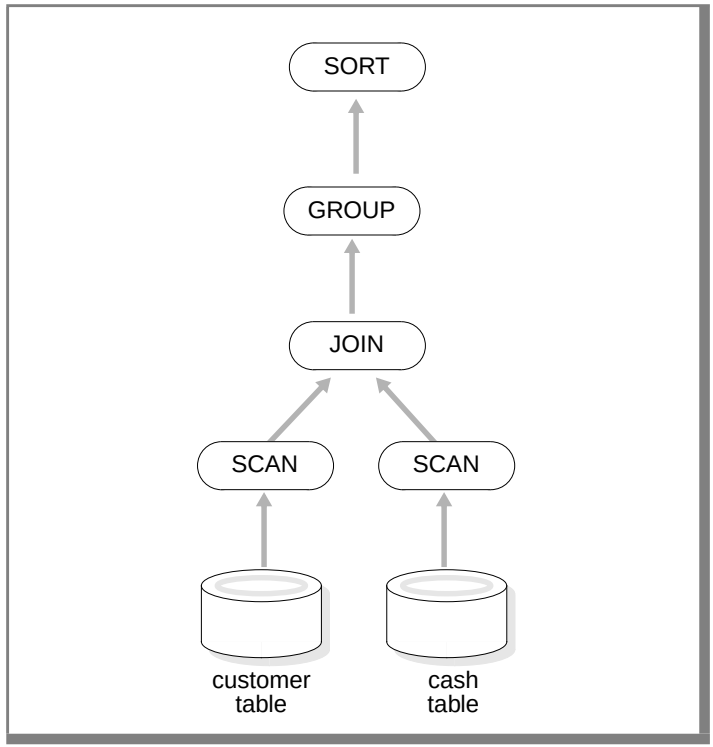


Figure 11-3
SQL Operator Plan for a Query

Exchanges

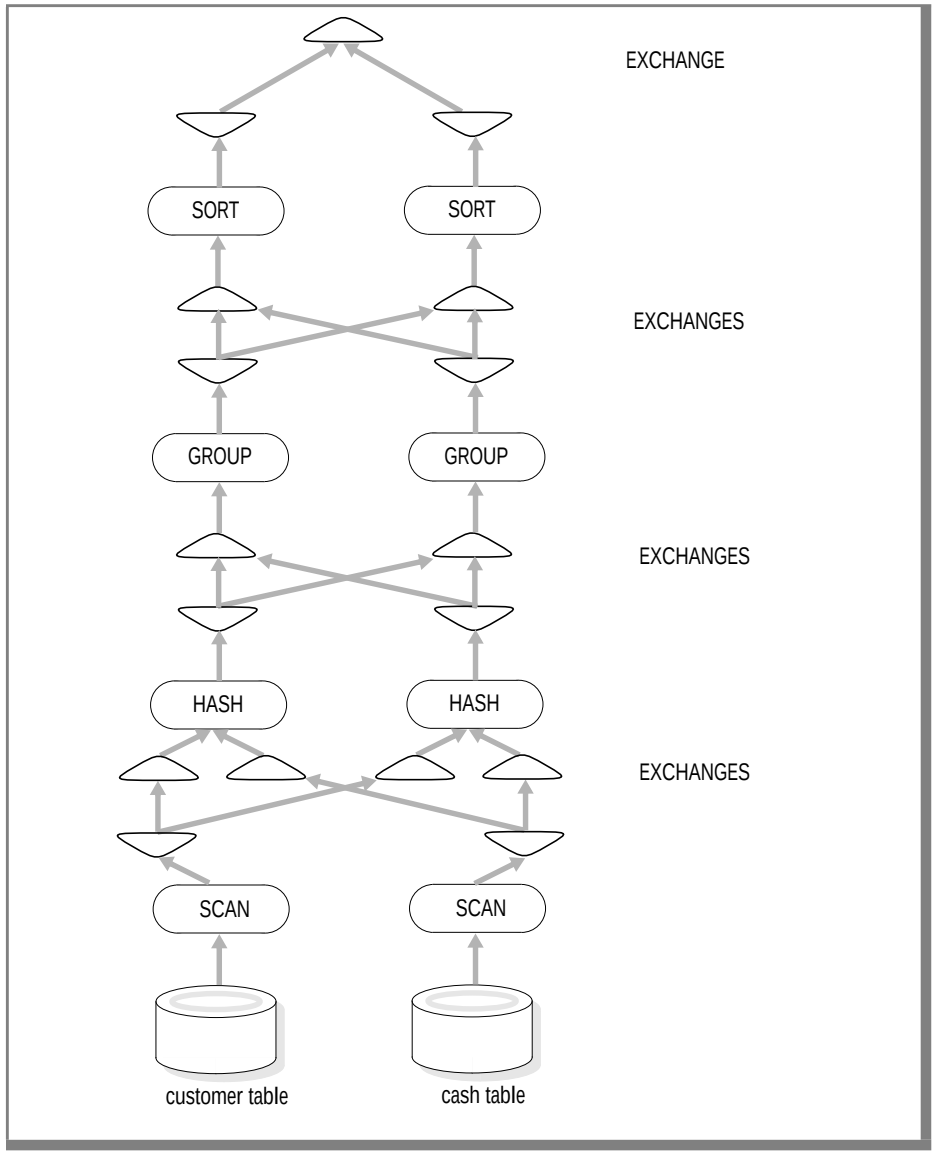
An *exchange* is an operator that serves as the boundary between SQL operator branches. An exchange facilitates parallelism, pipelining, and communication of data from producer instances of the SQL operator branch below it to consumer instances of the SQL operator branch above it. The optimizer inserts exchanges at places in an SQL operator plan where parallelism is beneficial.

When several producers supply data to a single consumer, the exchange coordinates the transfer of data from those producers to the consumer. For example, if a fragmented table is to be sorted, the database server usually creates a separate scan thread for each fragment. Because of different I/O characteristics, the scan threads might complete their work at different times. The optimizer uses an exchange to divide the data that the various scan threads produce into one or more sort threads with minimum buffering.

Depending on the complexity of the query, the optimizer might call for a multilayered hierarchy of producers, exchanges, and consumers.

Figure 11-4 shows how an exchange can add parallelism to the set of SQL operators from Figure 11-3 on page 11-9.

Figure 11-4
SImplified Illustration of Exchanges in SQL Operator Plan for a Query



The example in Figure 11-4 on page 11-11 shows how a single coserver with two CPU VPs and one fragment of each of the **customer** and **cash** tables might process the query. Starting from the bottom, Figure 11-4 shows the following operations:

- One CPU VPs scans a table fragment of the **customer** table while the other CPU VP scans a table fragment of the **cash** table.
- Exchange operators repartition the rows based on the join key and distribute the request to form a join with the repartitioned data to the two CPU VPs.

  The exchange operators coordinate data from both the **customer** and **cash** tables to ensure that **customer** and **cash** rows with the same key go to the same CPU VPs for the join operation.
- Two join operators use a hash-join method to combine the data from the two table fragments.
- Exchange operators receive the results of the joins and divide the rows for the GROUP operation.
- GROUP operators group the data by geographic area.
- Exchange operators combine the results of the GROUP operation and distribute the data to the sort instances.
- The SORT operator sorts the data by dollar sum.
- A final exchange operator combines the results of the sort.

On a real database server, of course, this example would be much more complicated. Many table fragments and many CPU VPs might exist, either on the same coserver or across many coservers. Figure 11-4 shows only the simplest case.

The database server can execute each instance of an SQL operator on a separate CPU VP, depending on the number of CPU VPs available. For example, if five coservers are configured with two CPU VPs on each coserver, the exchange operators can initiate 10 hash-join operators to increase the parallel execution of the joins.

Parallel Processing Threads

Depending on the resources that are available for a decision-support query, the Resource Grant Manager (RGM) assigns the different branch instances in a query subplan to different coservers. The optimizer initiates the query plan and the following threads:

- An **sqlexec** thread on the connection coserver

  The **sqlexec** threads manages the global session context on the coserver to which the session is connected.

- An **x_exec** thread on each participating coserver

  The **x_exec** threads manage the session context on the participating coservers.

- Additional threads to execute the branch instances

  The SET EXPLAIN output lists these threads as *secondary threads*.

The optimizer creates these secondary threads and exchanges automatically and transparently. They are terminated automatically as they complete processing for a given query. The optimizer creates new threads and exchanges as needed for subsequent queries.

Some monitoring tools display only the SQL operator but not the exchanges. For more information, refer to the "Monitoring Query Resource Use" on page 12-18.

Balanced Workload

The database server provides *small-table broadcast* to balance the workload automatically during the execution of a query.

During a hash join, either the hash table or probe table might be very small. A very small table is approximately 128 kilobytes or less in size plus an additional approximately 100 kilobytes of overhead so that it can fit into one hash partition. If the optimizer detects that either the hash table or probe table is very small, the optimizer decides to broadcast the small table within the hash partition across all coservers so that SQL operators can process data locally.

For example, the optimizer decides to execute the following SQL statement with small-table broadcast because the hash table is very small:

```
SELECT tab1.item FROM tab1, tab2
WHERE tab1.item = tab2.item AND tab2.num = 50
```

Figure 11-5 shows an excerpt from a sample **sqexplain.out** file that displays `Build Outer Broadcast` for the hash table.

```
QUERY:
------
select tab1.item from tab1, tab2
where tab1.item = tab2.item and tab2.num = 50

Estimated Cost: 2
Estimated # of Rows Returned: 9

1) virginia.tab1: SEQUENTIAL SCAN

2) virginia.tab2: SEQUENTIAL SCAN

   Filters: virginia.tab2.num = 50


DYNAMIC HASH JOIN (Build Outer Broadcast)
   Dynamic Hash Filters: virginia.tab1.item = virginia.tab2.item
```

Figure 11-5 *Sample sqexplain.out File for Small Table Broadcast*

Important: *To ensure that the optimizer has an accurate count of the number of table rows, run UPDATE STATISTICS in at least LOW mode on the tables that queries frequently join. If UPDATE STATISTICS has not been run or if the optimizer estimates that the table might be larger than 128 kilobytes, the database server does not attempt to use small table broadcast.*

Optimizer Use of Parallel Processing

In Extended Parallel Server, queries are always executed in parallel when the database operation requires data that is fragmented across multiple dbspaces and when multiple CPU VPs are available. Parallel query processing can occur on both a single coserver and across multiple coservers:

- Single coserver execution

  The database server executes database operations in parallel under the following circumstances:

  - The involved tables are fragmented across separate dbspaces on separate disks that are local to one coserver.
  - An SQL operator in the query plan processes a large amount of data so that it dynamically allocates multiple threads to execute in parallel across available CPU VPs on the single local coserver.

- Multiple coserver execution

  The database server achieves a high degree of parallelism for a query when the involved tables are fragmented across more than one coserver and multiple CPU VPs on the multiple coservers execute the SQL operators in the query plan.

The following database operations are executed in parallel when the involved tables are fragmented into separate dbspaces:

- DSS queries, which are usually complex SELECT statements that involve many rows to scan, join, aggregate, sort, or group
- INSERT, DELETE, and UPDATE statements that process nonlocal data
- Sorts
- Index builds
- UPDATE STATISTICS
- **onutil** CHECK TABLE DATA and CHECK INDEX command options
- Uncorrelated subqueries

The following sections describe how the database server uses parallel processing to execute these database operations.

Decision-Support Query Processing

The complex queries that are typical of DSS applications benefit from parallel processing. For example, queries that pose the following questions use parallel processing:

- Based on the predicted number of housing starts, the known age of existing houses, and the observed roofing choices for houses in different areas and price ranges, what roofing materials should we order for each of our regional distribution centers?
- How does the cost of health-care plan X compare with the cost of health-care plan Y, considering the demographic profile of our company? Would plan X be better for some regions and plan Y for others?

DSS applications perform complex tasks that often include scans of entire tables, manipulation of large amounts of data, multiple joins, and creation of temporary tables. Such operations involve many I/O operations, many calculations, and large amounts of memory.

Decision-support queries consume large quantities of non-CPU resources, particularly large amounts of memory. The optimizer usually allocates large amounts of memory to one or more of the following SQL operators:

- HASH JOIN
- SORT
- FLEX JOIN
- ANTIJOIN
- GROUP

For more information on how the optimizer uses SQL operators for parallel execution, refer to "Structure of Query Execution" on page 11-5.

Other factors also influence how the optimizer allocates resources to a query. Consider the following SELECT statement:

```
SELECT col1, col2 FROM table1 ORDER BY col1
```

If no indexes exist on **table1**, a sort is required, and the optimizer must allocate memory and temporary disk space to sort the query. However, if column **col1** is indexed, the optimizer can sometimes provide the order that the query specifies without consuming non-CPU resources. The optimizer might not choose to sort the table but instead retrieve rows in order by using the index. For more information on when to create indexes, refer to "Using Indexes" on page 13-13.

Parallel Data Manipulation Statements

DELETE, INSERT, and UPDATE statements perform the following two steps:

1. Fetch the qualifying rows.
2. Perform the delete, insert, or update.

The optimizer processes the first step of the statement in parallel, with one exception. The optimizer does not process the first part of a DELETE statement in parallel if the targeted table has a referential constraint that can cascade to a child table.

Parallel Inserts into Temporary Tables

The optimizer performs the following types of inserts in parallel:

- SELECT...INTO TEMP inserts that use explicit temporary tables
- INSERT INTO...SELECT inserts that use explicit temporary tables
- SELECT... INTO EXTERNAL TABLE statements that unload data to an external table

For more information on implicit and explicit temporary tables, refer to "Using Temporary Tables" on page 6-7 and the *Informix Guide to SQL: Syntax*.

The following sections explain the details and restrictions that apply to parallel inserts.

Explicit Inserts with SELECT...INTO TEMP

The optimizer can insert rows in parallel into explicit temporary tables that you specify in SQL statements of the form SELECT....INTO TEMP or SELECT...INTO SCRATCH. For example, the optimizer can perform the inserts in parallel into the temporary table, **temp_table**, as the following sample SQL statement shows:

```
SELECT * FROM table1 INTO SCRATCH temp_table
```

To perform the insert in parallel, the optimizer first creates a fragmented temporary table. The optimizer performs the parallel insert by writing in parallel to each of the fragments in a round-robin fashion. Performance generally improves as the number of fragments increases.

To obtain parallel inserts into fragments of a temporary table

1. Create a dbslice or set of dbspaces for the exclusive use of temporary tables and sort files.

   Use the **onutil** CREATE TEMP DBSLICE or **onutil** CREATE TEMP DBSPACE command to create this temporary space.

2. Set the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable to the dbslice or a list of two or more dbspaces that you created in step 1. If you set the DBSPACETEMP configuration parameter, you must restart the database server for it to take effect.

3. Execute the SELECT...INTO statement.

For more information about performance considerations for temporary space, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

Explicit Inserts with INSERT INTO.SELECT

The database server can also insert rows in parallel into explicit tables that the user creates for SQL statements of the form INSERT INTO...SELECT. For example, the database server processes the following INSERT statement in parallel:

```
INSERT INTO target_table SELECT * FROM source_table
```

The target table can be either a permanent table or a temporary table.

The database server processes this type of INSERT statement in parallel only when the in the following circumstances:

- The target table is fragmented into two or more dbspaces.
- The target table has no enabled referential constraints.
- Multiple CPU VPs are available on either of the following configurations:
  - On a single-coserver configuration, the NUMCPUVPS configuration parameter must be greater than 1.
  - On a multiple-coserver configuration, the NUMCPUVPS configuration parameter can be set to 1.
- The target table does not contain filtering constraints.
- The target table does not contain columns of TEXT or BYTE data type.

Parallel Index Builds

The database server can build indexes in parallel. For index builds the optimizer performs both scans and sorts in parallel. The following operations start index builds:

- CREATE INDEX
- Add a unique, primary key
- Add a referential constraint
- ALTER FRAGMENT

Parallel execution can occur during index builds when any of the following conditions exist:

- These operations involve multiple fragments.
- Multiple CPU VPs are available.
- Multiple sort threads are available.

If your coservers have multiple CPUs, the optimizer uses one sort thread per CPU VP during index builds.

The PDQPRIORITY setting controls the amount of sorting memory for index builds. To override the value of the PDQPRIORITY configuration parameter, set the **PDQPRIORITY** environment variable, or issue the SQL statement SET PDQPRIORITY.

For more information about index builds, refer to "Improving Performance for Index Builds" on page 7-22.

Parallel Processing and SPL Routines

In certain circumstances, the database server can use parallel execution for SQL statements in an SPL routine.

SQL Statements That Contain a Call to an SPL Routine

If an SQL statement contains a call to an SPL routine, the parts of the statement that are related to the SPL routine call are executed only by the connection coserver. SQL statements that are contained entirely in SPL routines might be executed in parallel, however.

The following sample query contains several calls to SPL routines:

```
SELECT y,x proc0(y)
   FROM tab1, tab2
      WHERE proc1(x) = proc2(y) AND x = proc3(x);
```

In this query, `SELECT y, x` and `FROM tab1, tab2` might be parallelized, but the WHERE clause is executed by the connection coserver. This restriction does not apply to execution of SQL statements that are contained in the SPL routines themselves, **proc1()**, **proc2()**, and **proc3()**.

SQL Statements in an SPL Routine

The optimizer can execute the SQL statements that are contained in an SPL: routine in parallel. The degree of parallelism depends on factors such as the SQL operators that make up the query plan for the individual SQL statement, the fragmentation strategy of the tables that are involved in each SQL statement, the number of CPU VPs available, and other resource availability. The query plans for the SPL routine SQL statements are cached so that they can be reused if the procedure is called again later in the session.

The PDQPRIORITY setting controls the amount of memory that a query can use. Queries specified with a low PDQPRIORITY value request proportionally smaller amounts of memory, so more of those queries can run simultaneously. To change the client value of PDQPRIORITY, embed the SET PDQPRIORITY statement in the body of the procedure.

Tip: *Informix suggests that you set PDQPRIORITY to `0` when you enter an SPL routine and then reset it for specific statements to avoid reserving large amounts of memory for the procedure and to make sure that the crucial parts of the procedure use the appropriate PDQPRIORITY setting.*

For example, the following procedure contains several SET PDQPRIORITY statements:

```
CREATE PROCEDURE my_proc (a INT, b INT, c INT)
       Returning INT, INT, INT;
SET PDQPRIORITY 0;
...
SET PDQPRIORITY 85;
SELECT ... (big, complicated, memory-consuming
             SELECT statement)
SET PDQPRIORITY 0;
...
;
```

Parallel Sorts

The optimizer can use parallel sorts for any query. Parallel sorts can occur both when the query executes on a single coserver and when the query executes across multiple coservers.

Query Execution on a Single Coserver

When a query executes on one coserver, the optimizer can allocate parallel sort threads when any of the following conditions occur:

- Data that the query requires is fragmented across multiple dbspaces that reside on different disks.
- Multiple processors are available for this query, and you specify multiple CPU VPs in the NUMCPUVPS configuration parameter.

- Adequate amounts of memory are available for the sort.
- Adequate amounts of temporary space are available if the sort overflows memory.

Estimating Sort Memory

The amount of virtual shared memory that the optimizer allocates for a sort depends on the number of rows to be sorted and the size of a row.

To calculate the amount of virtual shared memory needed for sorting

1. Estimate the maximum number of sorts that might occur concurrently.
2. Multiply this maximum number by the average number of rows and the average row size.

   For example, if you estimate that 30 sorts might occur concurrently, the average row size is 200 bytes, and the average number of rows in a table is 400, estimate the amount of shared memory that the optimizer needs for sorting as follows:

   ```
   30 sorts * 200 bytes * 400 rows = 2,400,000 bytes
   ```

Increasing Sort Memory

PDQPRIORITY specifies the amount of memory that a query can use for all purposes, including sorting.

When PDQPRIORITY is `0`, the maximum amount of shared memory that the optimizer allocates for a query is about 128 kilobytes for each SQL operator instance. To request more memory, set PDQPRIORITY to a value that is greater than `0`. The PDQPRIORITY setting requests a percentage of the amount of memory specified by the DS_TOTAL_MEMORY configuration parameter. The RGM divides the allocated memory evenly among the sort threads for the query.

Use one of the following methods to specify PDQPRIORITY:

- Set the PDQPRIORITY configuration parameter in the ONCONFIG file.
- Set the **PDQPRIORITY** environment variable.
- Execute the SQL statement SET PDQPRIORITY before you issue a query from a client program.

When the RGM controls the resources for a query, sorting does not use more than the amount of memory that is requested by the setting of PDQPRIORITY and limited by the setting of MAX_PDQPRIORITY.

Specifying Multiple Temporary Dbspaces

If your applications use temporary tables or large sort operations, you can improve performance by using the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable to designate a dbslice or one or more dbspaces for temporary tables and sort files.

For more information, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

Query Execution on Multiple Coservers

In Extended Parallel Server, parallel sorts occur even when PDQPRIORITY is not set. If the data to be sorted resides on different coservers, the optimizer creates one sort thread for each CPU VP.

The optimizer can use parallel sorts for any query when the following conditions occur:

- The data to be sorted resides in dbspaces on different disks across multiple coservers.
- You specify a multiple number of CPU VPs.

  If you specify a value of `1` for the NUMCPUVPS configuration parameter, but you have multiple coservers configured on different nodes, the optimizer can execute one sort thread per coserver.
- You specify enough memory to perform the sort.

  The setting of PDQPRIORITY determines the amount of memory available for a sort. When PDQPRIORITY is `0`, the maximum amount of shared memory that the optimizer allocates for a sort is about 128 kilobytes per sort instance. This amount of sort memory enables one sort thread per coserver.

  For information about the amount of memory per coserver when PDQPRIORITY is greater than `0`, refer to "Increasing Sort Memory" on page 11-22.

Other Sort Operations

Parallel sorts are not limited to queries with a PDQPRIORITY value greater than 0. Other database operations also use parallel sorts.

When PDQPRIORITY is greater than 0, queries and other SQL statements with sort operations benefit both from additional parallel sorts and from additional memory. These other SQL statements with sort operations include nonfragmented index builds and detached index builds.

For more information, refer to "Parallel Execution of UPDATE STATISTICS," which follows, and to "Parallel Processing and SPL Routines" on page 11-20.

Parallel Execution of UPDATE STATISTICS

The database server executes the SQL statement UPDATE STATISTICS in parallel. The optimizer uses the SQL operators SCAN and SORT to divide the processing for the UPDATE STATISTICS statement:

- When you execute UPDATE STATISTICS in HIGH mode, the optimizer scans the entire table. If the table is fragmented across multiple dbspaces on different disks, parallel scans can occur.
- When you execute UPDATE STATISTICS in MEDIUM or HIGH mode, the optimizer sorts the data to obtain data distributions. These sorts are not executed in parallel.

The setting of PDQPRIORITY affects the amount of memory that the UPDATE STATISTICS statement uses.

For more information on environment variables, refer to the *Informix Guide to SQL: Reference*. For more information on the syntax of SQL statements, refer to the *Informix Guide to SQL: Syntax*

Parallel Execution of onutil Commands

The database server can execute **onutil** commands in parallel, including the following commands:

- **onutil** CHECK TABLE DATA
- **onutil** CHECK INDEX
- **onutil** CREATE DBSLICE
- **onutil** ALTER DBSLICE...ADD DBSPACE

If the **onutil** command requires work on more than one coserver, it can be executed in parallel.

For information about the **onutil** utility, refer to the *Administrator's Reference*.

Correlated and Uncorrelated Subqueries

The optimizer can execute correlated and uncorrelated subqueries in parallel.

Parallel Execution of Nested Correlated Subqueries

A correlated subquery is a nested subquery whose WHERE clause refers to an attribute of a relation declared in the outer query. The optimizer evaluates the subquery for each row or combination of rows that the outer query returns.

The optimizer can unnest most correlated subqueries if the rewritten query provides a lower cost. For information about how the optimizer unnests correlated subqueries, refer to "Query Plans for Subqueries" on page 10-16.

If the optimizer cannot unnest a a correlated subquery, the optimizer can speed execution of the query with parallel execution.

Both the outer query and the correlated subquery can take advantage of parallel execution. The database sever uses the SQL operator EXPRESSION to process the subquery and can use parallel execution for the other SQL operators in the query plan, such as SCAN.

For example, suppose you have the following correlated subquery:

```
SELECT ... FROM tab1, tab2
   WHERE tab1.b = tab2.b
      AND tab1.a = ( SELECT tab3.a
                     FROM tab3
                     WHERE tab1.d = tab3.d) ...
```

Figure 11-6 shows how the query plan might look for this sample correlated query. The SQL operator EXPRESSION evaluates the subquery on tables **tab1** and **tab3**. If **tab1** and **tab2** are fragmented across multiple dbspaces, the optimizer can create multiple instances of the SQL operators SCAN and JOIN to enable parallel execution.

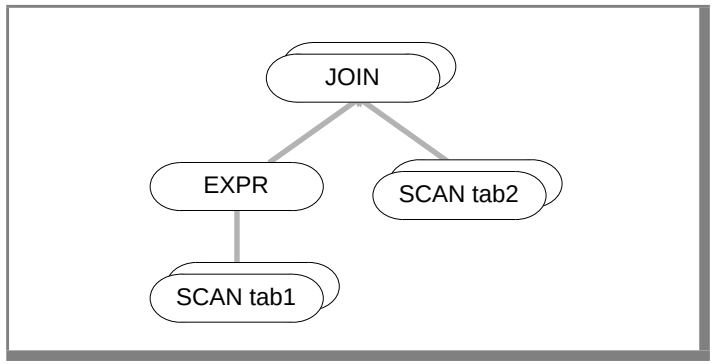


Figure 11-6
Sample Query Plan for Correlated Subquery

Parallel Execution of Uncorrelated Subqueries

An uncorrelated subquery is a subquery whose WHERE clause does not depend on a value that is returned in the main query. The optimizer executes an uncorrelated subquery only once. For uncorrelated subqueries, only the first thread that makes the request actually executes the subquery. Other threads then use the results of the subquery and can do so in parallel.

SQL Operations That Are Not Processed in Parallel

The optimizer does not process the following types of queries or statements in parallel:

- Queries started with an isolation mode of Cursor Stability

  Subsequent changes to the isolation mode do not affect the parallelism of queries already prepared. This situation results from the nature of parallel scans, which scan several rows simultaneously.

- OLTP queries that require quick response and return only a small amount of information

  The following section, "Processing OLTP Queries," describes considerations for processing OLTP queries.

- An UPDATE statement that has an *update* trigger that updates in the For Each Row section of the trigger definition

- Execution of data definition language (DDL) statements, such as CREATE DATABASE, CREATE TRIGGER, CREATE VIEW, and some ALTER TABLE statements

  For a complete list of DDL statements, see the *Informix Guide to SQL: Syntax*.

Important: *The optimizer executes the CREATE INDEX statement in parallel. For more information on building indexes in parallel, refer to* *"Parallel Index Builds" on page 11-19*.

Processing OLTP Queries

Not all queries should be processed in parallel. Although DSS applications can benefit from parallel processing, OLTP queries and other small queries often do not. The database server considers queries that require rows from only one table fragment to be OLTP queries. The optimizer recognizes OLTP queries and processes them appropriately.

Queries that do not use parallel processing require quick response and generate only a small amount of information. For example, the following queries do not use parallel processing:

- Do we have a hotel room available in Berlin on December 8?
- Does the store in Mill Valley have green tennis shoes in size 4?

For more information on DSS and OLTP applications, refer to "Decision Support" on page 1-9.

Chapter

12

Resource Grant Manager

In This Chapter

This chapter explains how the Resource Grant Manager (RGM) coordinates the use of resources for queries that are processed in parallel and how it decides which queries to run if more than one query is in the queue.

The chapter also describes:

- when and how to set the configuration parameters and environment variables that affect RGM resource coordination.
- how to monitor large, memory-consuming queries, with examples and explanation of command-line utility output.

Coordinating Use of Resources

The RGM dynamically allocates the following resources for DSS queries and other parallel database operations, such as building indexes:

- The amount of memory that the query can reserve in the virtual portion of database server shared memory

  The RGM uses values that you set to determine how to grant memory to a decision-support query. For information about adjusting the resources granted to decision support queries, refer to "Managing Resources for DSS and OLTP Applications" on page 12-11.

- The number of parallel threads that can be started for each query

  The RGM uses the SQL operators that make up the query plan to determine the number of threads to start for a query.

  For JOIN, GROUP, and SORT SQL operators, the RGM uses the following factors to determine the number of threads to start:

  - The values of configuration parameters (NUMCPUVPS, DS_TOTAL_MEMORY, DBSPACETEMP, and so forth) that the database server administrator sets

    For more information on setting the parameters that affect parallel processing, refer to "Controlling Parallel-Processing Resources" on page 12-12.
  - The number of CPU VPs available
  - The availability of computer-system resources (CPUs, memory, and disk I/O)

  For SCAN and INSERT SQL operators, the RGM also uses the number of fragments that the database operation accesses to determine the number of scan and insert threads (disk I/O).

For more information on fragmentation strategy guidelines to improve performance, refer to Chapter 9, "Fragmentation Guidelines." For more information on SQL operators, refer to "Structure of Query Execution" on page 11-5.

Important: *The RGM does not coordinate local queries or simple inserts, deletes, and updates if they execute on a single branch instance.*

If your database server has heavy OLTP use and you find performance is degrading, use the RGM-related parameters to limit the resources committed to decision-support queries. During off-peak hours, you can reserve a larger proportion of the resources for parallel processing, which achieves higher throughput for decision-support queries. For example, you might reduce the setting of DS_TOTAL_MEMORY to a smaller percent of SHMTOTAL. For additional information, refer to "Managing Resources for DSS and OLTP Applications" on page 12-11.

How the RGM Grants Memory

The RGM grants memory to a query for such operations as sorts, hash joins, and processing of GROUP BY clauses. The setting of DS_TOTAL_MEMORY determines the amount of memory that the RGM controls. The total amount of memory that all large memory-consuming queries use cannot exceed DS_TOTAL_MEMORY.

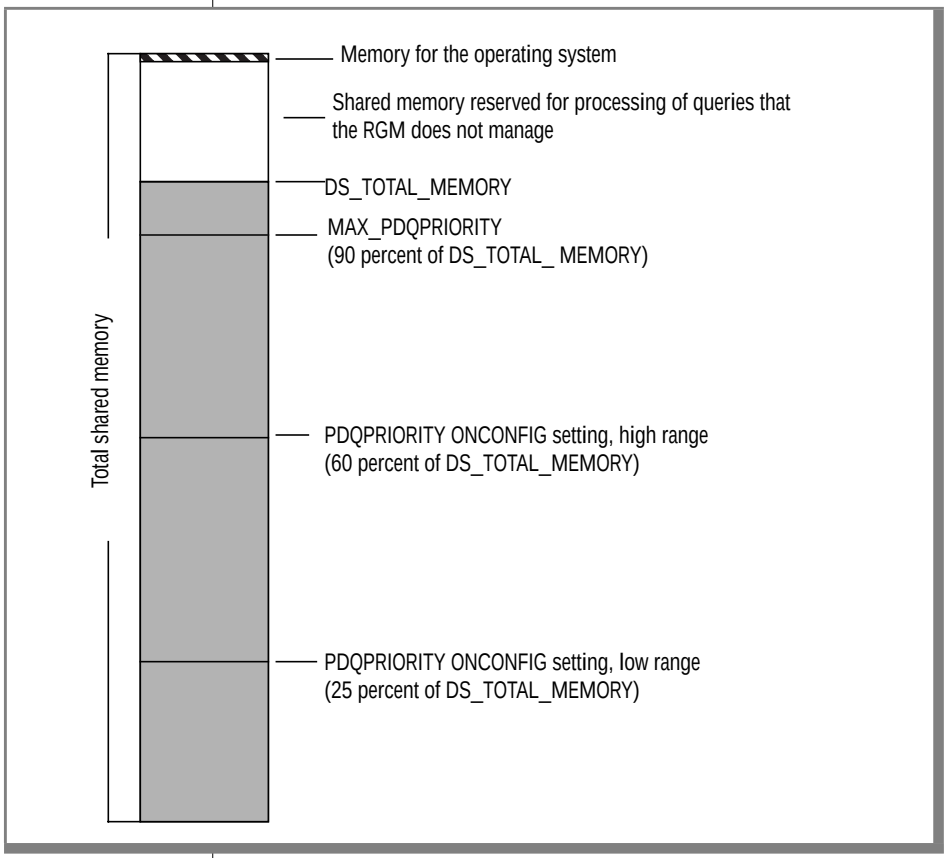


Figure 12-1
Sample RGM-Related Memory-Availability Configuration for a Single Coserver

The memory-availability configuration for a single coserver shown in Figure 12-1 provides about 15 percent of memory for OLTP and other application processing. It reserves about 85 percent for processing the queries that the RGM manages because these queries require large amounts of memory and use tables that are fragmented across coservers. The systemwide default setting of PDQPRIORITY in Figure 12-1 controls use of this memory. Although this systemwide default recommends that a single query be allocated a maximum and minimum amount of memory, users and applications can use the **PDQPRIORITY** environment variable and SQL statements to override the system default and request more or less memory.

Setting MAX_PDQPRIORITY to 90 provides a scaling factor that ensures no query can use more than 90 percent of the DS_TOTAL_MEMORY amount. In fact, the PDQPRIORITY amount requested by any query is scaled to 90 percent of the request. For example, a query run with the default PDQPRIORITY setting of 60 high and 25 low would be allocated a maximum of 90 percent of its 60 percent request, or 54 percent of DS_TOTAL_MEMORY.

A query can use the SQL statement SET PDQPRIORITY to request a single percentage of memory or a minimum and maximum percentage range of memory. The default set in the ONCONFIG file or by the application can be used instead if it is appropriate.

Although many system factors influence the amount of memory that the RGM grants to a single memory-consuming query, in general, the RGM uses the following formula to estimate the minimum amount of memory to grant to a single query:

```
min_memory_grant = DS_TOTAL_MEMORY*(min_pdq_priority/100)
                 * (MAX_PDQPRIORITY / 100)
                 * number_of_coservers
```

The value of *number_of_coservers* is the number of currently running coservers that your ONCONFIG file defines.

The value of *min_pdq_priority* is the integer value specified with the LOW keyword in the PDQPRORITY configuration parameter, the **PDQPRIORITY** environment variable, or the SQL statement SET PDQPRIORITY.

The most recent setting of PDQPRIORITY determines its value for a query. If the session sets PDQPRIORITY in the environment variable, that setting takes precedence over the ONCONFIG setting. If the query sets PDQPRIORITY with the SQL statement, that setting takes precedence over previous settings.

Use the following formula to estimate the maximum amount of memory that the RGM grants to a query:

```
max_memory_grant = DS_TOTAL_MEMORY*(max_pdq_priority/100)
                 * (MAX_PDQPRIORITY / 100)
                 * number_of_coservers
```

The value of *max_pdq_priority* is the maximum integer value specified with the HIGH keyword in the PDQPRORITY configuration parameter, the **PDQPRIORITY** environment variable, or the SQL statement SET PDQPRORITY.

For more information about using this environment variable and the SQL statement, refer to "Requesting Memory" on page 12-12.

Scheduling Queries

The RGM uses the following values to determine how to schedule a query:

- The integer that you specify in the SET SCHEDULE LEVEL statement
- The keyword that you specify in the DS_ADM_POLICY configuration parameter

After the RGM selects a query as the candidate query, the memory request specified in PDQPRIORITY determines whether the query can run immediately. If the required memory is not available, the query must wait. The table on page 12-10 provides an example of how the schedule level and the requested memory interact.

Setting Scheduling Levels

Users can set a scheduling level for a query to indicate its relative importance. In general, the RGM selects the query that has the highest scheduling level as the candidate query, although the admission policy determines which query actually executes next.

Because scheduling level indicates relative importance, the effect of a scheduling-level value depends on the scheduling levels of the other queries that are waiting for execution. If all queries have the same scheduling level, the admission policy and the amount of memory available determine the query-execution priority.

Use the SQL statement SET SCHEDULE LEVEL to set the scheduling level for a query. The default scheduling level is 50.

Using the Admission Policy

The RGM uses the admission policy to help determine query-scheduling order and whether or not to permit starvation. *Starvation* occurs when a query is delayed indefinitely behind other queries at a higher scheduling level.

The DS_ADM_POLICY configuration parameter determines whether the admission policy is STRICT or FAIR.

STRICT Policy

With the STRICT policy, the query with the highest scheduling level is the candidate query. If more than one query is at that level, the RGM selects the one with the earliest arrival time. Queries at the same level are executed in order of arrival if enough memory is available.

The STRICT policy allows query starvation to occur because a constant stream of high-priority queries can indefinitely delay a query with a lower-priority scheduling level.

FAIR Policy

With the FAIR policy, the RGM defines a fairness value for each waiting query. The fairness value takes into account the following factors:

- Scheduling level (assigned importance)
- How long the query has been waiting for execution
- PDQPRIORITY setting

The query that has been least fairly treated is the candidate query.

The RGM calculates the fairness value for a query by multiplying the time that a query has waited by its scheduling level. For example, if its requested memory is available, a query with a scheduling level of 20 that has waited sixty seconds runs before a query with a scheduling level of 40 that has waited only twenty seconds. Thus, this policy avoids starvation.

For queries that have the same fairness value, RGM tends to choose queries with lower PDQPRIORITY settings instead of those with higher PDQPRIORITY settings so that more queries can run simultaneously, but the FAIR policy also avoids starvation of higher PDQPRIORITY queries.

Specifying the Admission Policy

The default admission policy is FAIR. With the FAIR policy, the RGM takes into account how long a query has been waiting. The RGM determines the next candidate query by the largest fairness value, as calculated by the following formula:

```
fairness = (sched_lvl / 100) * wait_time
```

sched_lvl is the value that you specify for the query in the SET SCHEDULE LEVEL statement in SQL. The default value is 50.

wait_time is the number of seconds that the query has been in the wait queue.

Figure 12-2 shows an excerpt from sample output for **onstat -g rgm** that shows three queries in the wait queue.

```
Resource Grant Manager (RGM)
============================

DS_ADM_POLICY:                      FAIR
DS_MAX_QUERIES:                     10
MAX_PDQPRIORITY:                    100
DS_TOTAL_MEMORY:                    16000 KB
Number of Coservers:                2
DS Total Memory Across Coservers:   32000 KB

...
RGM Wait Queue:   (len = 3)
---------------
Lvl   Session     Plan    PdqPrio   Local Cosvr   Candidate   Wait Time
100    1.16        6       80-100                              1
40     1.18        7       50-70                               10
20     1.12        4       10-20                               20
...
```

Figure 12-2
Wait Queue Excerpt from onstat -g rgm Output

The RGM obtains the following fairness values for these three queries.

Query Plan	Fairness Value	PDQPRIORITY Range
6	100 / 100 * 1 = 1	80 to 100
7	40 / 100 * 10 = 4	50 to 70
4	20 / 100 * 20 = 4	10 to 20

Queries 7 and 4 have the same fairness values. However, the lower PDQPRIORITY setting of query 4 requests less memory, so query 4 is the candidate query that the RGM executes next.

Understanding the Effect of PDQPRIORITY on Query Admission

After the RGM chooses a candidate query, it determines whether it can execute the query.

If the minimum PDQPRIORITY resource request for the query is available, the RGM executes the query after setting its PDQPRIORITY to the largest available percentage within the range that PDQPRIORITY specifies.

If the minimum PDQPRIORITY setting for the query exceeds the available percentage of system resources, the RGM does not execute the query until enough resources are available.

Processing Local Queries

A *local query* is a query that accesses only tables on the connection coserver, which is the coserver to which the client is connected. The RGM grants local queries the PDQPRIORITY percent of the DS_TOTAL_MEMORY on the local coserver only, as the following formula shows:

```
memory_grant_local = DS_TOTAL_MEMORY*(min_pdq_priority/100)
                   * (MAX_PDQPRIORITY / 100)
```

Thus, when PDQPRIORITY is 100 for a local query, the RGM grants 100 percent of the specified DS_TOTAL_MEMORY shared memory on the local coserver.

You can use **onstat -g rgm** to display information about queries in the wait queue. The sample **onstat -g rgm** output in Figure 12-3 on page 12-19 shows that plan ID 2 in session 1.13 is an active local query.

Managing Must-Execute Queries

Must-execute queries are queries that the RGM allows to execute to prevent blocking. Must-execute queries include cursors, subqueries, triggers, and SPL routines

When the RGM executes these must-execute queries, the requested amount of memory might not be available. If the memory-grant amount, as calculated with the preceding formula, is not available, the RGM grants the query whatever memory is available or 128 kilobytes per SQL-operator instance, whichever is greater. The query is still executed, but it takes longer than when the requested amount of memory is allocated.

When the RGM does not grant the requested amount of memory to a must-execute query, an asterisk (*) appears next to the memory grant for the query in the Active Queue section of the **onstat -g rgm** output. For an example of a must-execute query, see plan ID 8 in session 1.15 in Figure 12-3 on page 12-19. Even though this query arrived after the other waiting queries, RGM allows it to execute because its PDQPRIORITY value is `0`.

Managing Resources for DSS and OLTP Applications

Systems that run both DSS and OLTP applications require careful control of resource use. Without resource management, the performance of decision-support applications might be uneven, and OLTP applications and other applications that share the system resources might perform badly.

A DSS application transaction is a query that the RGM manages if:

- it requires data from tables that are fragmented across coservers.
- it requires significant amounts of memory for joins, sorts, and so on.

Controlling Parallel-Processing Resources

This section describes how you can manage the configuration parameters that determine how the RGM executes queries. For general information about how the RGM uses these parameters, see "Coordinating Use of Resources" on page 12-3. For specific information about setting PDQPRIORITY, MAX_PDQPRIORITY, and DS_TOTAL_MEMORY, refer to Chapter 3, "Effect of Configuration on CPU Use."

Requesting Memory

PDQPRIORITY specifies the percentage of shared memory that the RGM can allocate for a query. If the PDQPRIORITY configuration parameter is not explicitly set, it has a value of 0, for which RGM grants a minimum of 128 kilobytes of shared memory for each SQL operator instance.

The database server administrator can set the PDQPRIORITY configuration parameter to specify a memory range as a minimum and maximum percentage of memory permitted for memory-consuming queries. To override this configuration parameter setting, the SQL user or database server administrator can set the PDQPRIORITY range with the SQL statement or the environment variable, as described in "PDQPRIORITY" on page 4-22.

The setting of MAX_PDQPRIORITY and DS_TOTAL_MEMORY limits the amount of memory each query is granted, as the following section describes.

The following sections describe how these different methods of requesting memory are interrelated and how the database server administrator can control the amount of memory allocated.

Assigning Memory Ranges to DSS Queries

PDQPRIORITY provides a minimum value and an optional maximum value for the percentage of shared memory that an individual query can use.

The largest PDQPRIORITY value in the range is the optimal memory allocation. The smallest PDQPRIORITY value is the minimum acceptable memory allocation for the query. The PDQPRIORITY range leaves the choice of the actual amount of allocated memory to the discretion of the RGM, and the choice depends on the current workload. Thus, the amount of memory allocated for a query can vary from one execution to another.

The query cannot run unless the minimum amount of memory is available. If you specify a single percentage of memory, the RGM can run a query only when the specified amount of memory is available. If you set a range of memory, the RGM can run a query when available memory falls in the specified range. The RGM ensures that the amount of memory granted to the query is significantly within the range requested by the setting of PDQPRIORITY.

For more information on how to limit memory that can be allocated to queries, see "Limiting the Memory That a Single DSS Query Uses" on page 12-14.

User Control of Resources

The PDQPRIORITY value can be any integer from -1 through 100. To override the PDQPRIORITY configuration parameter value for a query, use one of the following methods:

- Set the **PDQPRIORITY** environment variable. When the **PDQPRIORITY** environment variable is set in the environment of a client application, it limits the percentage of shared memory that can be allocated to any query that the client starts.
- Execute the SQL statement SET PDQPRIORITY statement before you issue a query from a client program.

  The values that the SET PDQPRIORITY statement set take precedence over the setting of the **PDQPRIORITY** environment variable or the PDQPRIORITY configuration parameter.

In effect, users can request a certain amount of memory for the query, but the setting of MAX_PDQPRIORITY limit the amount of memory that is actually allocated. For more details, refer to "Limiting the Memory That a Single DSS Query Uses" on page 12-14.

For example, you might execute the following SET PDQPRIORITY statement before you run a query:

```
SET PDQPRIORITY LOW 20 HIGH 50
```

This request allows the RGM to run the query if only 20 percent of the memory allowed to memory-consuming queries is available. If 30 percent of allowed memory is available, the RGM runs the query with the higher amount. Thus, the actual amount of memory allocated to a query can vary from one execution to another. If you specify only one memory percentage, the RGM runs the query only when the specified amount of memory is available.

Limiting the Memory That a Single DSS Query Uses

If your database server executes many memory-consuming queries at the same time, you might quickly exhaust the amount of memory that you specified in the DS_TOTAL_MEMORY configuration parameter. The active PDQPRIORITY setting and the MAX_PDQPRIORITY configuration parameter together determine the amount of DS_TOTAL_MEMORY memory to allocate to a query for parallel processing. Set these values correctly to ensure effective execution of DSS queries.

No well-defined rules exist for choosing these environment variable and parameter values. To get the best performance from your database server, take the following steps:

- Choose reasonable values for the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter.
- Observe the behavior of the database server and monitor query performance.
- Adjust these values.

The following sections discuss strategies for setting PDQPRIORITY and MAX_PDQPRIORITY for specific needs.

Maximizing DSS Use of Memory

To allow decision-support queries to use more memory, increase the value of MAX_PDQPRIORITY. If MAX_PDQPRIORITY is set too low, decision-support queries do not perform well.

If you want the database server to devote as much memory as possible to processing single DSS queries, set PDQPRIORITY and MAX_PDQPRIORITY to `100`.

If your system supports a multiuser query environment, you might set MAX_PDQPRIORITY to a low number to increase the number of queries that can run simultaneously at the cost of some query performance. Because queries compete for the same memory, a trade-off exists between running several queries more slowly and running only a single query very fast. As a compromise, you might set MAX_PDQPRIORITY to a low value such as `20` or `30`. With this setting, no query can be granted more than 20 or 30 percent of DS_TOTAL_MEMORY even if the query sets PDQPRIORITY to `100`, and five or more queries can run simultaneously.

Maximizing OLTP Throughput

At times, you might want to allocate memory to maximize the throughput of individual OLTP queries rather than allocate memory for decision support and parallel processing in queries. You can maximize OLTP throughput in one of the following ways:

- Reduce DS_TOTAL_MEMORY.

  Queries that require large amounts of memory cannot be allocated more memory than the value that you specify in the DS_TOTAL_MEMORY configuration parameter.

- Reduce DS_MAX_QUERIES.

  The DS_MAX_QUERIES configuration parameter specifies the maximum number of memory-consuming queries that can execute concurrently.

- Set MAX_PDQPRIORITY to a low value to reserve more memory for OLTP queries.

  When you set MAX_PDQPRIORITY to `0`, the amount of memory available for parallel execution is limited to 128 kilobytes for each SQL operator instance.

- If applications make little use of queries that require parallel sorts and parallel joins, consider setting PDQPRIORITY to a low value.

When you reduce the value of MAX_PDQPRIORITY or DS_TOTAL_MEMORY, response for decision-support queries is slow.

Adjusting Total Memory for DSS Queries

For information about the basic process for estimating the amount of shared memory to make available for decision-support queries, see "DS_TOTAL_MEMORY" on page 4-15.

To monitor memory that the RGM allocates to an executing query, run **onstat -g rgm**. This command displays only the amount of memory that is currently in use; it does not display the amount of memory that has been granted. For more information about this command, including output examples, refer to "Monitoring Query Resource Use" on page 12-18.

If you expect your database server to run OLTP applications at the same time that it runs DSS queries, use operating-system tools to monitor paging and swapping. When paging increases, decrease the value of DS_TOTAL_MEMORY so that the OLTP transactions can be processed quickly.

The value for DS_TOTAL_MEMORY that is derived from the following formula serves as a starting point for estimating the amount of shared memory to allocate to decision-support queries:

```
DS_TOTAL_MEMORY = p_mem - os_mem - rsdnt_mem - (128K * users)
            - other_mem
```

The variables in the formula are defined as follows.

Variable	Description
p_mem	Total physical memory available on host
os_mem	Size of operating system, including buffer cache
resdnt_mem	Size of Informix resident shared memory
users	Number of expected users (connections) specified by the third argument of the NETTYPE configuration parameter
other_mem	Size of memory used for other (non-Informix) applications

Many system factors influence the amount of memory granted to a single large memory-consuming query. For more information refer to "Limiting the Memory That a Single DSS Query Uses" on page 12-14. The minimum memory grant is 128 kilobytes for each SQL operator.

Limiting the Maximum Number of Queries

The DS_MAX_QUERIES configuration parameter limits the number of concurrent decision-support queries that can run. To estimate the number of decision-support queries that the database server can run concurrently, count each query that runs with PDQPRIORITY greater than 1 as one full query.

Because less memory is allocated to queries that run with lower PDQPRIORITY settings, you might assign lower-priority queries a PDQPRIORITY value between 1 and 30, depending on the resource impact of the query. If you use the setting of DS_MAX_QUERIES to control the number of DSS queries that run simultaneously, remember that the total number of queries that run with PDQPRIORITY values greater than 0 cannot exceed DS_MAX_QUERIES.

The PDQPRIORITY and MAX_PDQPRIORITY settings also limit the number of queries that run simultaneously because queries can run only if the requested amounts of memory are available, as explained in "Limiting the Memory That a Single DSS Query Uses" on page 12-14.

Changing Resource Limits Temporarily

User **informix** and user **root** can use the **xctl onmode** command-line utility to change the values of the following configuration parameters temporarily on all coservers:

- Use **xctl onmode -M** to change the value of DS_TOTAL_MEMORY.
- Use **xctl onmode -Q** to change the value of DS_MAX_QUERIES.
- Use **xctl onmode -D** to change the value of MAX_PDQPRIORITY.

These changes remain in effect only as long as the database server is up and running. The next time that the database server is initialized, it uses the values set in the ONCONFIG file.

For more information about these configuration parameters, refer to "Controlling Parallel-Processing Resources" on page 12-12. For more information about how to use **onmode** with **xctl**, refer to the *Administrator's Guide*.

If you change the values of the decision-support parameters regularly, such as to set MAX_PDQPRIORITY to `100` each night for processing reports, you can use a scheduled operating-system job to set the values. For information about creating scheduled jobs, refer to your operating-system manuals.

Monitoring Query Resource Use

Monitor the shared memory and thread resources that the RGM has granted for queries.

Monitor query resource use in any of these ways:

- For overall information about current queries in the system, run the **onstat -g rgm** utility, which the following sections describe. Examples of **onstat -g rgm** appear on page 12-19 and page 12-20.
- To write the query plan to an output file, execute a SET EXPLAIN statement before you run a query. For more information, refer to "Using SET EXPLAIN to Analyze Query Execution" on page 12-24.
- To capture information about specific aspects of a running query, run **onstat** utility commands. For more information on the uses of these commands and their output, refer to "Using Command-Line Utilities to Monitor Queries" on page 12-30.

For detailed information about the RGM and query optimization, refer to "Coordinating Use of Resources" on page 12-3.

Monitoring Queries That Access Data Across Multiple Coservers

To monitor current memory use across multiple coservers and to determine if any queries are in the wait queue, execute the **onstat -g rgm** option on **coserver 1**. This option reads shared-memory structures and provides statistics that are accurate at the instant that the command executes.

Figure 12-3 shows the first section of **onstat -g rgm** output.

Important: *You must issue the* ***onstat -g rgm*** *command from* ***coserver 1****. If you issue this command on another coserver, a message explains that the information is available only from* ***coserver 1****.*

```
Resource Grant Manager (RGM)
============================

DS_ADM_POLICY:                      FAIR
DS_MAX_QUERIES:                     10
MAX_PDQPRIORITY:                    100
DS_TOTAL_MEMORY:                    16000 KB
Number of Coservers:                2
DS Total Memory Across Coservers:   32000 KB
```

Figure 12-3
First Section of onstat -g rgm Output

The **onstat -g rgm** output in Figure 12-3 displays the values of the configuration parameters that affect parallel processing and the RGM calculated values.

Field Name	Field Description
DS_ADM_POLICY	Displays how the RGM schedules queries.
DS_MAX_QUERIES	Displays maximum number of memory-consuming queries that can be active at any one time on the database server.
DS_TOTAL_MEMORY	Displays maximum amount of memory that can be granted for use by memory-consuming queries on each coserver.

(1 of 2)

Field Name	Field Description
Number of Coservers	Displays number of coservers defined in your ONCONFIG file that are currently initialized.
DS Total Memory Across Coservers	Displays total amount of memory that is available across all coservers. The RGM calculates this value with the following formula: `Number of Coservers * DS_TOTAL_MEMORY`

(2 of 2)

The second section of the display, shown in Figure 12-4, describes internal control information for the RGM. It includes two groups of information.

```
...
Queries:    Waiting     Active
            4           3

Memory:     Total       Free
(KB)        32000       512
```

Figure 12-4
Second Section of onstat -g rgm Output

The first row is labelled **Queries** and contains the following information.

Column Name	Description
Active	Displays the number of memory-consuming queries that are currently executing.
Waiting	Displays the number of user queries that are ready to run but whose execution the database server deferred for admission-control reasons.

The next row is labelled **Memory** and displays the following information.

Column Name	Memory Section Column Description
Total	Displays the kilobytes of memory available for use by queries that require parallel processing. To calculate this total, RGM multiplies the DS_TOTAL_MEMORY value by the number of coservers defined in your ONCONFIG file that are currently initialized.
Free	Displays kilobytes of memory for queries that is not currently granted.

```
...
RGM Wait Queue:   (len = 4)
---------------
Lvl   Session     Plan    PdqPrio   Local Cosvr   Candidate   Wait Time
50    1.5         5       80-80                   *           54
50    1.16        6       80-100                              37
50    1.18        7       20-20                               35
10    1.12        4       10-20                               64
```

Figure 12-5
Third Section of onstat -g rgm Output

The third section of the display (**Wait Queue**), shown in Figure 12-5, describes the current RGM wait queue. It shows the queries that are waiting to be executed

Column Name	Description
Lvl	Displays the scheduling level for the query.
Session	Displays the global session ID for the coserver that initiated the query. The global session ID has the following format: *coserver_number.local-id* In this example, *coserver_number* is the connection coserver number, and *local-id* is the session ID on the connection coserver. In Figure 12-3 on page 12-19, the connection coserver for the first session in the **Wait Queue** is 1, and the session ID is 5.
PdqPrio	Displays the PDQPRIORITY range requested for the query.

(1 of 2)

Column Name	Description
Local Cosvr	For local queries, displays the coserver on which the query requires memory.
Candidate	Displays an asterisk (*) for a query that is waiting for sufficient memory.
Wait Time	Displays the number of seconds the query has been waiting. Queries can be waiting for one of the following reasons: ■ Not enough memory is available. ■ The number of active queries has reached the value for DS_MAX_QUERIES.

(2 of 2)

Figure 12-6 shows the last section of **onstat -g rgm** output.

```
...
RGM Active Queue: (len = 3)
-----------------
Lvl    Session      Plan    PdqPrio     Memory (KB)    #Cosvrs    Local Cosvr
70     1.14         3       90-100      28800          2
50     1.13         2       10-10       1600           1          1
50     1.15         8        0-0        *128           2
```

Figure 12-6
Last Section of onstat -g rgm Output

The last section of the display (**Active Queue**), which is shown in Figure 12-6, describes the current RGM active queues. This section shows the resources that are granted to each query.

Column Name	Description
Lvl	Displays the scheduling level for the query.
Session	Displays the global session ID for the coserver that initiated the query, in the following format: *connection_coserver.local-id* The *connection_coserver* is the number of the coserver where the user is logged in for this session. The *local-id* is the unique session ID on the connection coserver. In Figure 12-3 on page 12-19, the connection coserver for the first active query is 1, and the session ID is 14.
Plan	Displays the plan ID for the query.
PdqPrio	Displays the PDQPRIORITY range assigned to the query.
Memory	Displays the number of kilobytes of memory currently granted to the query. The sample **onstat -g xqs** output in Figure 12-16 on page 12-42 shows the total memory that is allocated to a query across coservers. If an asterisk (*) appears next to the memory amount, the session has been granted less than the requested amount of memory.
#Cosvrs	Shows the number of coservers that are executing the query. Figure 12-3 on page 12-19 shows that two coservers are executing the active query with global session ID 1.14.
Local Cosvr	For local queries, displays the coserver on which memory is allocated for the query.

Monitoring RGM Resources on a Single Coserver

To view query memory use for individual coservers, use the **onstat -g rgm csr** command. Figure 12-7 shows an example of this single coserver output. This **onstat -g rgm csr** output shows that memory on **coserver 1** is overallocated by 128 kilobytes, but 4000 kilobytes of memory is unused on **coserver 2**.

```
Resource Grant Manager (RGM)
============================

Per-Coserver Memory Information:

Cosvr   Total (KB)   Free (KB)   Over (KB)
1       8000                     128
2       8000         4000
```

Figure 12-7
onstat -g rgm csr Output for Local Query

Using SET EXPLAIN to Analyze Query Execution

The query optimizer decides how to perform a query and formulates a *query plan* for fetching the data rows that are required. A *query plan* states the order in which the database server examines tables and the methods by which it accesses the data in these tables to process a query. For more information on factors that affect the query-plan formulation, refer to "Filter Selectivity Evaluation" on page 10-22.

Figure 12-8 shows a simple query plan example on a single coserver system. Figure 12-15 on page 12-40 shows part of the output of a complex example with parallel processing across coservers.

Figure 12-8
Output from the SET EXPLAIN ON Statement

```
QUERY:
------
SELECT C.customer_num, O.order_num, SUM (I.total_price)
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
   GROUP BY C.customer_num, O.order_num;

Estimated Cost: 102
Estimated # of Rows Returned: 12
Temporary Files Required For: GROUP BY

1) pubs.o: SEQUENTIAL SCAN

2) pubs.c: INDEX PATH

 (1) Index Keys: customer_num (Key-Only)
 Lower Index Filter: pubs.c.customer_num = pubs.o.customer_num

3) pubs.i: INDEX PATH

 (1) Index Keys: order_num
 Lower Index Filter: pubs.i.order_num = pubs.o.order_num
```

After the query statement, the query plan includes the following information:

- The optimizer estimate of the work to be done in the units that it uses to compare plans

  Figure 12-8 shows that the cost for the query is 102. These units represent a relative time for query execution, in which each unit is assumed to be roughly equivalent to a typical disk access. The optimizer chose this query plan because the estimated cost for its execution was the lowest among all the evaluated plans.

- The optimizer estimate for the number of rows that the query is expected to produce, based on the information in the system catalog tables

  The example in Figure 12-8 on page 12-25 shows that the number of rows returned is estimated to be 12.

- The order in which tables are accessed and the method, or access path, that was used to read each table

  The plan in Figure 12-8 shows that the database server will perform the following actions:

  1. The database server reads the **orders** table first. Because no filter exists on the **orders** table, the database server must read all rows. Reading the table in physical order is the least expensive approach.
  2. For each row of **orders**, the database server searches for matching rows in the **customer** table. The search uses the index on **customer_num**. The notation `Key-Only` means that only the index need be read for the **customer** table because only the **c.customer_num** column is used in the join and the output, and that column is an index key.

     `Lower Index Filter` shows the key value where the index read begins. If the filter condition contains more than one value, an `Upper Index Filter` would be shown for the key value where the index read stops.
  3. For each row of **orders** that has a matching **customer_num**, the database server searches for a match in the **items** table using the index on **order_num**.

The following sections explain how to interpret information in the SET EXPLAIN output when you run a query such as the following one:

```
SET PDQPRIORITY 10;
SET EXPLAIN ON;
select geo_id, sum(dollars)
   from customer a, cash_rr b
   where a.cust_id=b.cust_id
   group by geo_id
   order by geo_id;
```

Displaying SQL Operator Statistics in the Query Plan

The SET EXPLAIN output provides the statistics for each SQL operator in a query plan. For a complete list of SQL operators and a description of the function of each operator, see "SQL Operators" on page 11-6.The following table lists the common SQL operators and explains how to interpret the information that appears.

SQL Operator	Statistics Reported
EXPRESSION	Number of rows evaluated and the coserver number for each branch instance
FLEX INSERT	Number of rows inserted into the temporary table and the coserver number for each branch instance
FLEX JOIN	Number of rows produced, number of rows in build, number of rows in probe, kilobytes of memory allocated, number of partitions written as overflow to temporary disk space, and the coserver number for each branch instance
GROUP	Number of rows produced, number of rows consumed, kilobytes of memory allocated, number of partitions that were written as overflow to temporary disk space, and the coserver number for each branch instance
HASH JOIN	Number of rows produced, number of rows in build, number of rows in probe, kilobytes of memory allocated, number of partitions that were written as overflow to temporary disk space, and the coserver number for each branch instance
NESTED LOOP JOIN	Number of rows produced and the coserver number for each branch instance
PROJECT	Number of rows produced and the coserver number for each branch instance
SCAN	Number of rows produced, number of rows scanned, and the coserver number for each branch instance
SORT	Number of rows produced and the coserver number for each branch instance

To display these statistics, use the following monitoring tools:

- The **sqexplain.out** file that results from the SQL statement SET EXPLAIN ON
- Output of **onstat -g xqs *qryid***

For more information on how the database server divides a query for execution, refer to "Structure of Query Execution" on page 11-5.

To display SQL operator statistics in the sqexplain.out file

1. Execute the SQL statement SET EXPLAIN ON from a client connection.
2. Execute the query.
3. Look at the statistics in **sqexplain.out** file.

Figure 12-9 shows statistics excerpts from a sample **sqexplain.out** file.

Figure 12-9
Statistics Excerpt from the sqexplain.out File

```
XMP Query Statistics

Cosvr_ID: 1
Plan_ID: 9

  type  segid brid information
  ----  ----- ---- -----------
  scan   6    0    inst cosvr time   rows_prod  rows_scan
                   ---- ----- ----   ---------  ---------
                   0    1     0      1          1
                   ------------------------------------
                   1                 1          1

  scan   7    0    inst cosvr time   rows_prod  rows_scan
                   ---- ----- ----   ---------  ---------
                   0    1     13     758        991161
                   1    1     12     687        989834
                   2    1     13     677        975101
                   3    1     12     691        972258
                   4    1     11     736        952424
                   5    1     12     686        981833
                   6    1     9      330        628271
                   7    1     8      359        641173
                   8    1     9      345        671366
                   9    1     5      402        343214
                   10   1     4      343        345530
                   11   1     5      359        334115
                   ------------------------------------
                   12                6373       8826280

  hjoin  5    0    inst cosvr time   rows_prod  rows_bld   rows_probe  mem        ovfl
                   ---- ----- ----   ---------  --------   ----------  ---        ----
                   0    1     13     0          0          0           80         0
                   1    1     39     6373       1          6373        88         0
                   2    1     13     0          0          0           80         0
                   3    1     13     0          0          0           80         0
                   ------------------------------------------------------------------
                   4                 6373       1          6373        (6656)
```

This **sqexplain.out** file shows the following query-plan information.

Column Name	Description
type	The SQL operator type
segid	The ID of the segment within a query plan that contains the operator
brid	The branch ID within the segment that contains the SQL operator
information	SQL operator-specific statistics, including the time for each instance of each operator

Adjusting PDQPRIORITY Settings to Prevent Memory Overflow

For increased query-processing efficiency, hash-join overflows to temporary space are read back in asynchronously through read-ahead buffers. These read-ahead buffers are either 16 kilobytes in size or the size of the largest row, whichever is larger, and the buffers are read with a light-scan operation. Nevertheless, try to avoid hash-join overflow to temporary space if possible.

If **onstat -g xqs** output or the **sqexplain.out** file shows that hash joins are overflowing to temporary space, you might adjust PDQPRIORITY to ensure that the query is granted enough memory.

For example, consider the following sample section of **onstat -g xqs** output:

```
hjoin  2    0    inst cosvr time   rows_prod  rows_bld   rows_probe  mem ovfl
-----  -   --    ---  ---  -------- ------     --------
               0    1     5        52           9995        52       1672    0
               1    2     7        48           60005       48       3048    1
-------------------------------------------------
               2                   100          70000       100     (4160)
```

The **mem** column shows the approximate number of kilobytes of memory required to fit all the build rows in coserver memory. If the number in the **ovfl** column is greater than 0, that instance had a memory overflow. The number in parentheses in the last row of the example is the total memory allocated for the HASH JOIN across all the coservers. If `-1` appears in the **ovfl** column, the operator did not build a hash table.

In this example, **coserver 1** requires approximately 1672 kilobytes for all the build rows and **coserver 2** requires 3048 kilobytes. However, `1` in the **ovfl** column for **coserver 2** indicates that the HASH JOIN on **coserver 2** did not get enough memory.

To calculate the PDQPRIORITY value that would give **coserver 2** the memory that it requires for this HASH JOIN, divide the memory required (3048 kilobytes) by the value of DS_TOTAL_MEM, which is 16,000 kilobytes in this case. Then multiply the result by 100.

If more than one memory-consuming instance overflows, ensure that all instances get enough memory by choosing the instance that has the largest memory requirement.

Using Command-Line Utilities to Monitor Queries

You can use various **onstat** options to determine how queries and transactions are being run in your database server. Although **onstat** information provides only a snapshot of the current state of the system, you can use the **-r** option to run it repeatedly at specified intervals. The following sections illustrate the most commonly used **onstat** options with examples.

For complete information about **onstat** options, see the *Administrator's Reference*.

The following table lists command-line utilities for specific resource-monitoring purposes. Run the command-line utilities from the connection coserver, which is the coserver where the query originated.

Command	Description
onstat -g rgm	Displays information about RGM parameters and policies, as well the queries in the wait and active queues. This option displays RGM wait and active queue information for each query currently in the system. Use this information to get a snapshot of current query status and resource use. Use the session ID to identify or request query information in other **onstat** command output. For an output sample and more information, refer to "Monitoring Queries That Access Data Across Multiple Coservers" on page 12-19.
onstat -g sql	Displays SQL statement information by session. For more information, refer to "Monitoring SQL Information by Session" on page 12-35.
xctl onstat -g xmp	Displays information about the query segments and SQL operators that are currently executing on each coserver. This information includes session IDs. For more information, refer to "Monitoring Query Segments and SQL Operators" on page 12-36.
onstat -g xqs *qryid*	Displays query statistics for a specific query plan. Obtain the value for *qryid* from the output of the **onstat -g rgm** or **onstat -g xmp** commands. SET EXPLAIN output also displays query-plan information, as discussed in "Using SET EXPLAIN to Analyze Query Execution" on page 12-24. For more information, refer to "Monitoring SQL Operator Statistics" on page 12-40.
onstat -g xqp *qryid*	Displays summary information about a specific query plan. Obtain the value for *qryid* from the output of the **onstat -g rgm** or **onstat -g xmp** commands. For more information, refer to "Monitoring Query Plans" on page 12-39.

(1 of 2)

Command	Description
onstat -g ses	Displays shared memory and threads that specific sessions use. For more information, refer to"Monitoring SQL Information by Session" on page 12-35.
xctl onstat -u and **xctl onstat -g ath**	Display user threads and transactions on all coservers, listed by coserver. For more information, refer to "Monitoring User Threads and Transactions" on page 12-43.
onstat -g dfm and **onstat -g xmf**	Display data flow between coservers on the high-speed interconnect. For more information, refer to "Monitoring Data Flow Between Coservers" on page 12-46.

(2 of 2)

Figure 12-10 shows the output of the **onstat -g ses command** on the connection coserver to display a list of sessions on all coservers.

Figure 12-10
onstat -g ses Output

```
local                                          #RSAM     total    used
sessid   sessid  user      tty      pid        hostname  threads  memory   memory
1.10     64      informix  -        436        -         1        65536    29208
1.63     63      informix  -        0          -         0        16384    8408
1.60     62      gxypl     -        8619       -         3        139264   111320
1.61     61      informix  -        0          -         0        16384    8408
1.60     60      gxypl     pts/2    8619       mesmero   1        98304    75520
1.56     59      informix  -        8618       -         1        81920    28920
1.54     57      superv    -        8587       -         2        147456   95960
1.56     56      informix  pts/1    8618       mesmero   1        73728    30704
1.54     54      superv    pts/4    8587       mesmero   1        98304    75192
1.10     10      informix  CEREBUS  436        cerebus   1        73728    30896
1.8      8       informix  -        0          -         0        16384    12368
1.7      7       informix  -        0          -         0        16384    8408
1.5      5       informix  -        0          -         0        16384    12368
1.4      4       informix  -        0          -         0        16384    8408
1.3      3       informix  -        0          -         0        16384    8408
1.2      2       informix  -                   -         1        24576    16864
```

The **onstat -g ses** option output displays the following information:

- The shared memory allocated for a session that is running a decision-support query
- The shared memory used by a session that is running a DSS query
- The number of threads that the database server started for a session

Use the **onstat -g ses** option to obtain this information for only the coserver on which you issue the **onstat** command.

For example, Figure 12-10 shows that session 1.60 has started four threads.

To display detailed information about a session, run the **xctl onstat -g ses *session id*** option. Figure 12-11 provides an example of the **onstat -g ses *session id*** output.

```
       local                                        #RSAM    total      used
sessid sessid   user         tty  pid      hostname threads memory     memory
1.81    81      informix     7    12914    port_mei 1       114688      59888

tid      name     rstcb    flags    curstk   status
2850     sqlexec_ 30ad8540 --BP---  1736     sleeping(secs: 1)

Memory pools    count 1
name         class addr     totalsize freesize #allocfrag #freefrag
81           V     31252010 114688    54800    307        10

name           free       used           name           free       used
overhead       0          104            scb            0          80
opentable      0          5704           filetable      0          856
log            0          4184           temprec        0          1608
ralloc         46304      16384          gentcb         0          8448
ostcb          0          2064           net            0          3704
sqscb          0          11736          rdahead        0          104
sqlparser      8312       0              hashfiletab    0          280
osenv          184        704            sqtcb          0          400
fragman        0          1048

Sess      SQL            Current            Iso Lock       SQL  ISAM F.E.
Id        Stmt type      Database           Lvl Mode       ERR  ERR  Vers
1.81      CREATE INDEX   idxtsterdb         CR  Not Wait   0    0    8.20

Current SQL statement :
  create index i on orders (order_num)

Last parsed SQL statement :
  create index i on orders (order_num)
```

Figure 12-11
onstat -g ses id Output

The **xctl onstat -g ses *session id*** option displays the threads and memory allocated for and used by a session. The **onstat -g ses *session id*** output in Figure 12-11 contains the following information.

Column Name	Column Description
Session ID and local session ID	The database server uses *global session numbers* in the following format: *connection_coserver.local-id* The *connection_coserver* is the number of the coserver where the user is logged in for this session. The *local-sessid* is the session ID on the connection coserver.
User	The login name of the connected user
Tty	The terminal ID
Pid	The UNIX process ID If the process ID is a negative number, the process is a client process running an external application. For example, user sessions that run ad hoc queries from applications on PCs will have negative PIDs.
Hostname	The name of the database server to which the session is attached
# RSAM threads	The number of threads executing transactions or queries for this session
Total Memory	The total amount of memory allocation for this session
Used memory	The amount of allocated memory that the session is using

Monitoring SQL Information by Session

Use the **xctl onstat -g sql *session-id*** option to display summary information about the last or currently executing SQL statement in the session.

The **onstat -g sql *session-id*** command displays the following information:

- Type of SQL statement
- Database name
- Isolation level
- Lock mode

Figure 12-12 provides an example of the **onstat -g sql** output.

```
Sess      SQL            Current          Iso Lock       SQL  ISAM F.E.
Id        Stmt type      Database         Lvl Mode       ERR  ERR  Vers
1.81       -              idxtsterdb       CR  Not Wait   0    0    8.30
1.81       CREATE INDEX   idxtsterdb       CR  Not Wait   0    0    8.30
```

Figure 12-12
Sample onstat -g sql Output for Connection Coserver

Monitoring Query Segments and SQL Operators

Use the **xctl onstat -g xmp** option to display information about the query segments and SQL operators that are currently executing on a coserver.

Figure 12-13 and Figure 12-20 show the XMP Query Segments section and the XMP Query Operators section of **onstat -g xmp** output on **coserver 1**.

```
XMP Query Segments
segid  width  numbr  qryid  sessid    flags      seqno
0      1      1      1644      1.123   0x11a      1
2      2      1      1644      1.123   0x118      3
1      2      1      1644      1.123   0x118      4
0      1      1      1656      1.87    0x11a      1
2      2      1      1656      1.87    0x118      3
4      2      1      1656      1.87    0x118      4
1      2      1      1656      1.87    0x118      5
0      1      1      1657      1.107   0x11a      1
2      2      1      1657      1.107   0x118      3
4      2      1      1657      1.107   0x118      4
```

Figure 12-13
onstat -g xmp Query Segments Section

Figure 12-13 shows the first section of **onstat -g xmp**, which contains Query Segments information.

Column Name	Description
segid	The ID of the segment within a plan
width	The number of instances for this branch for the entire plan, not just this coserver
numbr	The branch ID within the segment
qryid	The plan ID Use this value for the **onstat -g xqp *qryid*** and **onstat -g xqs *qryid*** commands.
sessid	The global session ID for the user
flags	The processing flags for the segment
seqno	The sequence number, which represents the order that the segment within the plan was activated for execution

The second section of the display shows information about the SQL operators that the query uses.

Figure 12-14
onstat -g xmp Query Operators Section

```
...
XMP Query Operators
opaddr     qry  segid branch brtid opname  phase  rows   in1        in2
0x111f5118 1644 2     0-0    5966  xchg    open   53436  0x111f52e0 0x0
0x111f52e0 1644 2     0-0    5966  nljoin  next   53579  0x111f54d8 0x111cdca0
0x111f54d8 1644 2     0-0    5966  xchg    next   244    0x0        0x0
0x111f5118 1644 2     0-0    5966  xchg    open   53436  0x111f52e0 0x0
0x111f52e0 1644 2     0-0    5966  nljoin  next   53579  0x111f54d8 0x111cdca0
0x111f54d8 1644 2     0-0    5966  xchg    next   244    0x0        0x0
0x111cdca0 1644 2     0-0    5966  scan    next   3584   0x0        0x0
0x111f7788 1644 1     0-0    5967  xchg    open   113704 0x111f7948 0x0
0x111f7948 1644 1     0-0    5967  antij   next   113827 0x111fd1c8 0x0
0x111fd1c8 1644 1     0-0    5967  xchg    next   113948 0x0        0x0
0xb458468  1656 2     0-0    6007  xchg    open   1560   0xf276020  0x0
0xf276020  1656 2     0-0    6007  hjoin   probe  1692   0xf2761c0  0xf276420
0xf2761c0  1656 2     0-0    6007  xchg    done   99     0x0        0x0
0xf276420  1656 2     0-0    6007  xchg    next   1709   0x0        0x0
0xf27b4f0  1656 4     0-0    6008  xchg    open   1708   0xc70e020  0x0
0xc70e020  1656 4     0-0    6008  scan    next   1817   0x0        0x0
0xed586b8  1656 1     0-0    6009  xchg    create 0      0xed58878  0x0
0xed58878  1656 1     0-0    6009  sort    create 0      0xb45bfb8  0x0
0xb45bfb8  1656 1     0-0    6009  xchg    next   4680   0x0        0x0
0xefcff10  1657 2     0-0    6011  xchg    create 0      0xefd0238  0x0
0xefd0238  1657 2     0-0    6011  group   create 0      0xefd0638  0x0
0xefd0638  1657 2     0-0    6011  hjoin   probe  1669   0xefd0a38  0xefd0ba8
0xefd0a38  1657 2     0-0    6011  xchg    done   99     0x0        0x0
0xefd0ba8  1657 2     0-0    6011  xchg    next   1669   0x0        0x0
0xf4be2f0  1657 4     0-0    6012  xchg    done   1668   0xf4be618  0x0
0xf4be618  1657 4     0-0    6012  scan    done   1898   0x0        0x0
```

Figure 12-14 shows the second section of **onstat -g xmp**, which contains Query Operators information.

Field Name	Description
opaddr	The in-memory address of operator structure This information helps to associate **in1** and **in2** values with other operators
qry	The plan ID for the query
segid	The ID for the segment within a plan
branch	The ID for the branch within the segment

(1 of 2)

Field Name	Description
brtid	The thread ID that is executing the branch instance
opname	The type of SQL operator
phase	The processing phase of SQL operator
rows	The number of rows that this SQL operator processed
in1/in2	The address of SQL operators Operators are constructed into tree structures, and **in1** and **in2** represent linkage operators. A `0x0` value represents no child.

(2 of 2)

For more information on query segments, refer to "Displaying SQL Operator Statistics in the Query Plan" on page 12-26.

Monitoring Query Plans

Use the **xctl onstat -g xqp *qryid*** option to display summary information about a specific query plan. A plan can be displayed only from the connection coserver, which is the coserver where the **sqlexec** thread is running.

The database server uses SQL operators and exchanges to divide a query plan into segments and construct a tree of operators. The **onstat -g xqp** command displays this operator tree in the order of execution. The order of the operators in the output of the **onstat -g xqp** command is the same as in the output for the SQL statement SET EXPLAIN ON. For more information on SQL operators and exchanges, refer to "Structure of Query Execution" on page 11-5. For more information on the SQL statement SET EXPLAIN ON, refer to "Using SET EXPLAIN to Analyze Query Execution" on page 12-24.

The following sample command displays information about query plan ID 9:

```
onstat -g xqp 9
```

Figure 12-15 shows the output for the preceding command.

```
XMP Query Plan

  oper       segid  brid   width
  ---------------------------------
  scan       5      0      2
  scan       6      0      2
  hjoin      4      0      1
  scan       7      0      1
  hjoin      3      0      1
  group      3      0      1
  group      2      0      1
  flxins     1      0      1
```

Figure 12-15
Sample onstat -g xqp Output for Query Plan 9

This **onstat -g xqp** display shows the following query-plan information.

Column Name	Description
oper	The type of SQL operator For more information on the types of SQL operators, refer to "Displaying SQL Operator Statistics in the Query Plan" on page 12-26 and "SQL Operators" on page 11-6.
segid	The ID of the segment within a plan that contains the operator
brid	The branch ID within the segment that contains the operator
width	The number of instances of this branch in the query plan Multiple instances can exist for each branch because the SQL operator can execute in parallel on multiple fragments of a table.

Monitoring SQL Operator Statistics

You can use the **onstat -g xqs *qryid*** option to display query statistics for a specific query plan. A plan is available for display only on the connection coserver, which is the coserver on which the **sqlexec** thread is running.

The database server uses SQL operators and exchanges to devise a query plan. The **onstat -g xqs** command displays this SQL operator tree in the order of execution. The information that appears for each SQL operator instance includes the following statistics:

- Type of SQL operator
- Coserver number
- Number of rows that each specific SQL operator processed
- Amount of memory granted to each specific SQL operator
- Amount of memory that each specific SQL operator used
- Number of partitions that were written to temporary disk space

The **onstat -g xqs** command displays statistics for all currently active query plans. The statistics reported by **onstat -g xqs** are updated periodically as the query runs, and might not reflect the current state of the query. Some information, such as overflow information, might not be useful until the query is complete. The final statistics for the query appear in the **sqexplain.out** file if you run SET EXPLAIN ON before you run the query.

The following sample command displays SQL operator statistics for query plan ID 4146:

```
onstat -g xqs 4146
```

Figure 12-16 shows the output for the preceding command.

```
Cosvr_ID: 1
 Plan_ID: 4146

  type  segid brid information
  ----  ----- ---- -----------
  scan   5    0    inst cosvr time   rows_prod  rows_scan
                   ---- ----- ----   ---------  ---------
                   0    1     1      5000       5000
                   1    1     1      4998       4998
                   ------------------------------------
                   2                 9998       9998

  scan   6    0    inst cosvr time   rows_prod  rows_scan
                   ---- ----- ----   ---------  ---------
                   0    1     59     200030     200030
                   1    1     158    800154     800154
                   ------------------------------------
                   2                 1000184    1000184

  hjoin  4    0    inst cosvr time   rows_prod  rows_bld   rows_probe  mem ovfl
                   ---- ----- ----   ---------  --------   ----------  --- ----
                   0    1     174    26725      9998       1000184     592 23
                  ------------------------------------------------------------
                   1                 26725      9998       1000184     (192)

  scan   7    0    inst cosvr time   rows_prod  rows_scan
                   ---- ----- ----   ---------  ---------
                   0    1     1      1000       1000
                   ------------------------------------
                   1                 1000       1000

  hjoin  3    0    inst cosvr time   rows_prod  rows_bld   rows_probe  mem ovfl
                   ---- ----- ----   ---------  --------   ----------  --- ----
                   0    1     172    2631       1000       26725       8    0
                  ------------------------------------------------------------
                   1                 2631       1000       26725       (1592)

  group  3    0    inst cosvr time   rows_prod  rows_cons   mem         ovfl
                   ---- ----- ----   ---------  ---------   ---         ----
                   0    1     172    2631       2631        104         0
                   ------------------------------------------------------
                   1                 2631       2631        (128)

  group  2    0    inst cosvr time   rows_prod  rows_cons   mem         ovfl
                   ---- ----- ----   ---------  ---------   ---         ----
                   0    1     5      2631       2631        216         2
                   ------------------------------------------------------
                   1                 2631       2631        (120)

  flxins 1    0    inst cosvr time   it_count
                   ---- ----- ----   --------
                   0    1     1      2632
                   -------------------------
                   1                 2632
```

Figure 12-16
Sample onstat -g xqs Output for Query Plan 4146

The **onstat -g xqs** output in Figure 12-16 shows the following SQL operator statistics.

Field Name	SQL Operator Statistics
type	The type of SQL operator
segid	The ID of the segment within a query plan that contains the operator
brid	The branch ID within the segment that contains the operator
information	SQL operator-specific statistics, including the time for each SQL operator instance, the number of kilobytes of memory required to build the hash table, and the number of partitions written as overflow to temporary space. If `-1` appears in the **Mem** or **Ovfl** columns, no hash table was built. For information about these SQL operator-specific statistics, refer to "Displaying SQL Operator Statistics in the Query Plan" on page 12-26.

The order of the operators in the output of the **onstat -g xqs** command is the same as in the output of the following tools:

- **onstat -g xmp** command

  For more information, refer to "Monitoring Query Segments and SQL Operators" on page 12-36.

- SQL statement SET EXPLAIN ON

  For more information, refer to "Using SET EXPLAIN to Analyze Query Execution" on page 12-24.

For more information on SQL operators and exchanges, refer to "Structure of Query Execution" on page 11-5.

Monitoring User Threads and Transactions

Each decision-support query has a primary thread on the connection coserver. This primary thread can start additional threads to perform tasks for the query, such as scans and sorts.

To obtain information on all the threads that are running for a decision-support query, use the **xctl onstat -u** and **xctl onstat -g ath** options.

The **xctl onstat -u** option lists all the threads that are executing queries. The primary thread and any additional threads on participating coservers are listed. If you issue the **onstat -u** command on a specific coserver, only the threads on that specific coserver appear in the output.

The thread information shows what threads each session is running, how busy each thread is, and how many locks each thread holds.

For example, session 10 in Figure 12-17 has a total of five threads running.

```
Userthreads
address  flags   sessid   user     tty      wait     tout locks nreads
nwrites
80eb8c   ---P--D 0        informix -        0        0    0     33       19
80ef18   ---P--F 0        informix -        0        0    0     0        0
80f2a4   ---P--B 3        informix -        0        0    0     0        0
80f630   ---P--D 0        informix -        0        0    0     0        0
80fd48   ---P--- 45       chrisw   ttyp3    0        0    1     573      237
810460   ------- 10       chrisw   ttyp2    0        0    1     1        0
810b78   ---PR-- 42       chrisw   ttyp3    0        0    1     595      243
810f04   Y------ 10       chrisw   ttyp2    beacf8   0    1     1        0
811290   ---P--- 47       chrisw   ttyp3    0        0    2     585      235
81161c   ---PR-- 46       chrisw   ttyp3    0        0    1     571      239
8119a8   Y------ 10       chrisw   ttyp2    a8a944   0    1     1        0
81244c   ---P--- 43       chrisw   ttyp3    0        0    2     588      230
8127d8   ----R-- 10       chrisw   ttyp2    0        0    1     1        0
812b64   ---P--- 10       chrisw   ttyp2    0        0    1     20       0
812ef0   ---PR-- 44       chrisw   ttyp3    0        0    1     587      227
 15 active, 20 total, 17 maximum concurrent
```

Figure 12-17
onstat -u Output

P in the fourth **flags** column indicates the primary thread for a session. The code in the first **flags** column indicates why a thread is waiting. The possible flag codes are as follows.

Flag Code	Event for Which Thread Is Waiting
B	Buffer
C	Checkpoint
G	Logical-log write
L	Lock

(1 of 2)

Flag Code	Event for Which Thread Is Waiting
S	Mutex
T	Transaction
Y	Condition
X	Rollback
DEFUNCT	The thread has incurred a serious assertion failure and has been suspended to allow other threads to continue their work. If this status flag appears, refer to the appendix in the *Administrator's Reference* that explains thread suspension.

(2 of 2)

The **onstat -g ath** option display also lists the primary thread and secondary threads on the connection coserver and participating coservers. In addition, it includes a **name** column that describes the activity of the thread, as Figure 12-18 shows.

Figure 12-18
onstat -g ath Output

```
Threads:
tid     tcb      rstcb    prty    status                 vp-class   name
5       40067294 0        4       sleeping(secs: 0)        1cpu     xmf_svc
12      407c85e8 0        2       running                  5aio     aio vp
38      40a0deec 4006bdd0 2       sleeping(secs: 49)       1cpu     btclean
40      409e5c84 4006c540 4       sleeping(secs: 1)        1cpu     onmode_mon
128129  40b97788 4006df48 2       sleeping(secs: 1)        1cpu     sqlexec_1.3567
128130  4138187c 4006f598 2       sleeping(Forever)        1cpu     x_exec_1.3567
128134  40b01d18 4006d420 2       ready                    1cpu     x_hjoin_0
128135  412266cc 40070478 2       sleeping(Forever)        1cpu     x_flxjoin_0
128136  408cd1a0 4006fd08 2       sleeping(Forever)        1cpu     x_scan_0
128137  408cd3cc 4006f950 2       running                  1cpu     x_scan_4
128138  904510d4 4006e6b8 2       sleeping(Forever)        1cpu     x_scan_8
```

Figure 12-18 output lists the following items:

- The primary thread, `sqlexec_1.3567`, on the connection coserver

  The session ID (1.3567) is part of the name of the **sqlexec** thread.

- The **x_exec** thread, `x_exec_1.3567`, that the primary thread starts on a participating coserver to carry the context of the **sqlexec** thread

  The **x_exec** thread initiates secondary threads on the participating coserver.

- Three *scan* threads, `x_scan_0, x_scan_4,` and `x_scan_8`, that either the **sqlexec** or **x_exec** thread starts, depending on which coserver the data resides
- A *hash-join* thread, `x_hjoin_0`, that starts as soon as two scan threads have data to join

Monitoring Data Flow Between Coservers

If the **onstat -g xqs** output shows data skew, use the **onstat -g dfm** and **onstat -g xmf** options to diagnose the problem. These options display work and overall cycle information, as well as data flow between coservers on the high-speed interconnect.

Tip: *Imbalance in data flow between coservers might result from operating-system or application activity on one of the coservers or across the high-speed interconnect.*.

To detect data skew that might be caused either by the fragmentation strategy of the tables or by a large number of duplicate values in joined columns in the query, use **onstat -g dfm** to monitor data flow between coservers.

```
 DFM Information    cosvr_id:    1
--------------------------------------
 max glob pk: 1000    num glob pk: 0

 delay: 10            q pk: 2             init cred: 100
 rsnd_fact: 30        rsnd_batch: 5       rsnd_bwait: 2
 maxq: 450            max qpk: 225        minq: 20
 maxq_rstep: 8        maxq_num_steps: 4

 l1_cong_fact: 3      l2_cong_fact: 10    l3_cong_fact:20
 l1_cong_rspnd: 2     l2_cong_rspnd: 5
 l1_cong_trspnd: 10   l2_cong_trspnd: 20
```

Figure 12-19
Global Packet Statistics inf onstat -g dfm Output

Look at the global-packet statistics in the first part of the **onstat -g dfm** output, which Figure 12-19 on page 12-46 shows. The following table lists each field in the global-packet section of **onstat -g dfm** output and explains how to interpret its statistics.

Field	Description
max glob pk	Value is determined by the number of CPU VPs in the system For one or two CPU VPs, the value of **max glob pk** is `1000`. For more than two and fewer than six CPU VPs, the value is `2000`. For more than six CPU VPs, the value is `3000`.
num glob pk	The current number of XMF (eXtended Message Facility) buffers for the database server A value of the **num glob pk** field that approaches the value in the **max glob pk** field and stays high for a sustained period of time might indicate a poor fragmentation strategy, data skew caused by the filters in the query, or insufficient memory. Insufficient memory causes overflow to temporary space and requires disk reads for processing. The SET EXPLAIN output field, **ovfl**, indicates if an operation is overflowing to temporary space. If `-1` appears in the **ovfl** field, the GROUP operator did not build a hash table.
delay	The number of milliseconds that the sender threads wait before resending a packet
init cred	Initial credits assigned The database server assigns 100 credits to each sender thread. Every time that the sender sends a packet, the sender loses one credit. When the receiver sends an acknowledgment back to the sender, the sender regains a credit.
rsnd_batch	The number of resends after which the sender thread waits longer than **delay** to resend a packet
rsnd_bwait	The factor by which **delay** is multiplied to specify the number of milliseconds that the sender thread waits after the **rsnd_batch** number of packet resends In the output sample in Figure 12-19, the sender thread waits `10 * 2` milliseconds after every fifth resend.
rsnd_fact	The actual resend delay, which depends on whether congestion is detected

(1 of 2)

Field	Description
q pk	The largest number of packets in each sender queue that the receiver can queue before enforcing flow control policies
max q pk	Limits the value of **q pk**
maxq	Maximum queue length
minq	Minimum queue length
Congestion fields	Congestion indicators Three levels of congestion specify increasing factors that delay senders when the database server detects congestion. These congestion-level factors provide flow control that smooths the flow of traffic across the interconnect. When congestion is detected, the actual sender delay is `delay * cong_fact`.

(2 of 2)

Figure 12-20 shows the **Sender information** and **Receiver information** sections in the second part of the **onstat -g dfm** output.

Figure 12-20
Sender and Receiver Information in onstat -g dfm output

```
DFM Information    cosvr_id:    1
--------------------------------------
 ...
 Sender information:
 xpl.seg.br.bi(tid): cred av: snd pend: msgs tot: rsnd q: rsnd tot: rsnd pend: rrsnd tot: rsnd skip:
 249.1.0.0(635)          0          1       1541      1       1224        1         447       2754

 253.2.0.0(645)          3          0       1526      1       0           0         0         0


  Receiver information:
    xpl.seg.br.bi(tid):   q len:    q total:    q disc:    q empty:      q seq:
     249.0.0.0(630)          449       2528        1771       3/593         0/0
         schid: 0    nsndr: 2           max_q:450             qpk_lim:4
           sender: 0        (debt) 100            (disc) 1
           sender: 1        (debt) 100            (disc) 1
     249.1.0.0(635)           0         3845        0          2403/2280     0/0
         schid: 1    nsndr: 2           max_q:450             qpk_lim:4
     253.0.0.0(640)           0         0           0          0/846         0/0
     253.2.0.0(645)           0         36          0          27/27         0/0
         schid: 2    nsndr: 2           max_q:450             qpk_lim:4
     253.2.0.0(645)           0         1334        0          193/137       0/0
         schid: 3    nsndr: 2           max_q:450             qpk_lim:4
     253.1.0.0(648)           0         3402        0          1765/1723     0/0
         schid: 1    nsndr: 2           max_q:450             qpk_lim:4
```

In the **Sender Information** and **Receiver Information** sections of the **onstat -g dfm** output, the **xpl.seg.br.bi(tid)** field corresponds to the following fields in the **onstat -g xmp** display.

Portion of Field	Description
xpl	The query ID, which is the same value as the **qry** field
seg	The segment ID, which is the same value as the **segid** field
br	The branch ID, which is the same value as the first digit in the **branch** field
bi	The branch instance, which is the same value as the second digit in the **branch** field
(tid)	The thread ID, which is the same value as the **brtid** field

Sender Information contains the following statistics for each thread.

Field	Description
cred av	The current number of available credits for this coserver to send messages A value of `0` in this **cred av** field indicates that the sender can no longer send packets until a receiver returns credits after processing the sent data. Although the value `0` might appear momentarily in **onstat -g dfm** output, the last credit is always returned to prevent deadlocks. A low value in the sender **cred av** field indicates that the receivers are not processing data and returning credits quickly, which might be a sign of data skew in the query.
snd pend	The number of messages waiting
msgs tot	The total number of messages sent
rsnd q	Resend statistics
rsnd tot **rsnd pend**	High values in the **rsnd q**, **rsnd tot**, or **rsnd pend** fields indicate DFM communications problems that might result from a receiver problem, such as a coserver down, poor fragmentation strategy, or data skew resulting from many duplicate values in the join column.
rrsnd tot	The number of times resent messages were sent again
rsnd skip	The number of times a message was not resent because of flow-control logic that delays messages when appropriate
Congestion statistics	The number of times congestion was detected (These values are `0` and are not shown in Figure 12-20.)

The **Receiver information** section contains the following statistics for each thread.

Field	Description
q len	The number of messages currently waiting to be processed
q total	The total number of messages that have been processed
q disc	The total number of messages that have been discarded
q empty	Empty queue information The first number in the field shows the number of times that the queue was empty when the message was queued. The second number in the field shows the number of times the receiver checked for a message and had to wait for one. These numbers tell you that the receiver thread was could have processed more work, perhaps because the sender is slow.
q seq	Packet sequence information The first number is the sequence number of the first packet in the queue. The second number is the sequence number of the last packet in the queue.

To see how well data is flowing for each SQL operator, correlate the query ID, segment ID, and branch ID to the **onstat -g xmp** output. For a sample of **onstat -g xmp** output, see Figure 12-13 on page 12-36 and Figure 12-20 on page 12-49.

Use **onstat -g xmf** to monitor the high-speed interconnect between coserver nodes.

```
XMF Information
---------------
  Cosvr_id: 1    Domain_Cnt: 1

  Poll Information:

    Domain_ID  Interval   Current    Average    Cycle      Wk_Cycles  In_DG/Sig
    ---------------------------------------------------------------------------
    0          10         10         10         621918498  113281       N / N

  ...

  Coserver Information:

    ID  X_Msgs  X_Bytes    R_Msgs  R_Bytes    X_Rtrns R_Dupls XOffs XO_Cycls
    ---------------------------------------------------------------------------
    1      1529    1109420      1529     1109420       0         0     0   0
    2      1244    323321     111658     1399806       3         4     0   0
    3      376     315612        334       57054      11       173     0   0
```

Figure 12-21
Selected onstat -g xmf Output

When you monitor the high-speed interconnect with **onstat -g xmf**, follow these general guidelines:

- Check the overall statistics in **Coserver Information**, which is the last section of the **onstat -g xmf** output.

  Coserver Information displays the following statistics by coserver. The values in these fields indicate whether or not traffic between coservers is balanced and not skewed toward any one coserver:

  - The **X_Msgs** and **X_Bytes** fields display the number of messages and bytes that each coserver transmits.
  - The **R_Msgs** and **R_Bytes** fields display the number of messages and bytes that each coserver receives.
  - The **X_Rtrns** field displays the number of retransmits to this coserver.
  - The **R_Dupls** field displays the number of duplicate messages that were received from this coserver

Because some high-speed interconnects have limited kernel buffer space for each connection, their buffer space might be exhausted when interconnect traffic is heavy. When the buffer space is exhausted, some packets are dropped and must be retransmitted.

If you see a large number of retransmits (**X_Rtrns**) with a low number of duplicate packets received (**R_Dupls**), you might adjust the setting of the SENDEPDS configuration parameter. It is usually not worth altering this parameter unless you see a large number of retransmits with a low number of duplicate packets received, which means that packets are being transmittied but are not arriving at the remote end. For more information about the SENDEPDS parameter, consult your machine notes file.

- Check **Poll Information** in the first part of the **onstat -g xmf** output:
  - The **Average** field displays the average poll-time interval.
  - The **Cycle** field displays the number of polls.
  - The **Wk_Cycles** field displays the number of polls that have resulted in work.

  The values in the **Average**, **Cycle**, and **Wk_Cycles** fields indicate how often the database server checks the interconnect without any work to do, as calculated by the following formula:

  ```
  percent_poll_work = Wk_Cycles / Cycle
  ```

  A low percentage by itself does not indicate a problem. However, if a query is taking a long time to complete, but the percentage of poll time that results in work is low, a problem exists somewhere. The problem might be that a coserver is down, a data skew exists, or the fragmentation strategy is incorrect. To locate the problem, check the status of the nodes and coservers, the statistics in the **onstat -g dfm** output, and other **onstat** utility output.

Chapter

13

Improving Query and Transaction Performance

In This Chapter

This chapter provides practical suggestions for improving the performance of individual queries and transactions. Information in this chapter includes tuning information for both DSS and OLTP applications and explains when and how tuning techniques are appropriate for large queries or for real-time transactions.

This chapter discusses the following topics:

- Evaluating and monitoring individual queries
- Possible query and transaction performance improvements:
  - Maintaining the statistical information that the optimizer uses to choose a query plan
  - Creating more useful indexes
  - Improving filter statements
  - Taking advantage of new SQL extensions
  - Reducing the effects of join and sort operations
  - Reviewing the optimization and isolation level

For conceptual and general background information, see Chapter 10, "Queries and the Query Optimizer," and Chapter 11, "Parallel Database Query Guidelines." For monitoring information, see Chapter 12, "Resource Grant Manager."

Evaluating Query Performance

Performance tuning is an iterative process. Each query and each database server is different, so you must use your judgement to evaluate the output of tuning utility programs and make tuning decisions. After you make tuning adjustments, re-evaluate the effect and make further adjustments if necessary.

Before you change a query, study its SET EXPLAIN output to determine the kind and amount of resources that it requires. The SET EXPLAIN output shows what parallel scans are used, the maximum number of threads required, the indexes used, and so on. "Using SET EXPLAIN to Analyze Query Execution" on page 12-24 provides an example of SET EXPLAIN output.

This section describes a general approach to evaluating a particular query in a DSS database server. The information comes from user experience with DSS applications that use Extended Parallel Server.

Tip: *Before you begin to evaluate queries, create a text file or printout that maps all of the coservers and disks on the database server. The file should contain all disk aliases, so that you can easily compare statistics that the operating system produces as well as output from the database server utilities. For quick reference when you are evaluating command-line utility output, you might also create a file that contains the names of all tables and the dbspaces or dbslices across which they are fragmented.*

Monitoring Query Execution

The major concerns in performance tuning for queries include:

- balance of processing across coservers.
- balance of resource use.

You can monitor individual queries in the following ways:

- Execute the SET EXPLAIN statement before you run a query to write the query plan to an output file.
- Use command-line utilities to create snapshot information about a query at a particular moment.

Depending on the kind of query analysis that you are performing, you might choose one or another of these methods. For full analysis, use both methods.

Because balance of processing across coservers and balanced resource use is a major concern in performance tuning for a query, you need to monitor process balance.

To detect skew during query processing

1. Examine repeated **onstat -g rgm** output to see information about pending queues and the basic memory allocation for the query.

   The **onstat -g rgm** output also displays the session and query plan IDs, which you use in analyzing output from other **onstat** options to track specific queries.

2. Check the **onstat -g ses** and **onstat -g sql** output to see basic session information, such as the SQL statement. Because you execute these **onstat** options on the connection coserver, you see information about all coserver processes.

3. Examine the output of **onstat -g xqp** and **onstat -g xqs**, which you execute on the connection coserver. These two **onstat** options display query segment information for the entire database server.

4. Examine the output of **xctl onstat -g xmp**, which displays query information by SQL operator on each coserver.

5. In addition, you might obtain CPU and disk I/O information from operating-system utilities, and you can run the following database server utility programs to examine processing across the high-speed interconnect between coservers:

   - Run **xctl onstat -g xps** and examine any signs of operators that use all of their allocated memory and write output to temporary space.

     If the query uses a lot of temporary space, run **xctl onstat -d** and examine the output to see how the space is being used.

   - Examine the output of **xctl onstat -g dfm** and **xctl onstat -g xmf** to see work and overall cycle information as well as sender and receiver information.

For more information on using these **onstat** options, including sample output, refer to"Monitoring Query Resource Use" on page 12-18. For general information about these and other **onstat** options, refer to the utilities chapter in the *Administrator's Reference*.

As you gain experience with query monitoring and tuning your database server, you can write script files that automate data collection by calling the utility programs that you find most useful.

Improving Query and Transaction Performance

Several user-controlled factors combine to affect query and transaction performance. The most important factors are as follows:

- Up-to-date distribution and table-size statistics

  Run UPDATE STATISTICS regularly and often. When the optimizer chooses the query plan, it uses the distribution and table-size statistics that are stored for each table in the system catalog tables. If these statistics do not reflect the current size and content of the table, the optimizer might not choose the most efficient plan.

  For information about refreshing distribution statistics, see "Maintaining Statistics for Data Distribution and Table Size" on page 13-8.

- Appropriate fragmentation

  Ideal fragmentation schemes allow the database server to eliminate table fragments that are not required to satisfy the query and to identify the fragments that contain the rows requested by a transaction.

  Although no single fragmentation scheme can provide this benefit for all queries and transactions, if you evaluate the queries and transactions that are run most often, you can usually create a fragmentation scheme that permits fragment elimination for many of them.

  For more information about fragmentation and fragmentation schemes, see Chapter 9, "Fragmentation Guidelines."

- Useful indexes

  If you create several indexes on a table and design each index for one of your most common queries or transactions, the optimizer is more likely to find an index that increases efficiency. In addition, the optimizer has more indexes from which to choose when it executes a query or transaction.

  Examine the queries that are run in your system, and determine the indexes that the optimizer might use for greater efficiency. Do not create more indexes than you are sure that you need. When tables are modified, the more indexes that must be updated, the longer it takes to complete the table modification.

  For more information about creating useful indexes, see "Using Indexes" on page 13-13.

- Efficient SQL code

  The SQL code in the query might sometimes force inefficient use of resources. The optimizer can correct some of these problems automatically. For example, the optimizer might unnest subqueries or create automatic indexes. Nevertheless, the more efficient you make the SQL code, the better the query performs. Certain kinds of expressions slow query performance, as described in "Improving Filter Selectivity" on page 13-27.

  In addition, the database server provides several SQL extensions that might improve specific queries and transactions. For information about these extensions, see "Using SQL Extensions for Increased Efficiency" on page 13-30.

In addition to these factors, how you use your database server also affects the performance of queries, especially queries that run simultaneously with other queries or transactions.

Database server use includes concurrency issues and the load on system resources caused by operating-system processes or other processes and applications that are not associated with the database server. Although discussion of these problems is outside the scope of this manual, monitoring the entire system use on your database server nodes can help you identify resource concurrency issues.

Maintaining Statistics for Data Distribution and Table Size

The optimizer uses the statistics in the system catalogs to determine the lowest-cost query plan. Make sure that you keep these statistics up-to-date so that the optimizer can choose the best query plan. Whenever tables are updated after they are created and populated, run UPDATE STATISTICS to ensure that statistics are also updated. If tables are updated by adding many rows in batch processes or by attaching table fragments, run UPDATE STATISTICS as part of the update process. If tables are updated by insertions, deletions, and updates of individual rows, run UPDATE STATISTICS on a regular schedule.

UPDATE STATISTICS uses parallel processing to run fast, as described in "Parallel Execution of UPDATE STATISTICS" on page 11-24. For information about the setting of BUFFERS and the speed of UPDATE STATISTICS, see "Tuning BUFFERS" on page 4-13.

The following sections provide guidelines for these topics:

- Updating the number of rows
- Creating data distributions
- Updating statistics for join columns
- Improving performance for the UPDATE STATISTICS statement

Updating the Number of Rows

When you run UPDATE STATISTICS LOW, the database server updates the statistics in the table, row, and page counts in the system catalog tables

LOW is the default mode for UPDATE STATISTICS. The following sample SQL statement updates the statistics in the **systables**, **syscolumns**, and **sysindexes** system catalog tables but does not update the data distributions:

```
UPDATE STATISTICS FOR TABLE tab1;
```

Run UPDATE STATISTICS LOW frequently for tables in which the number of rows changes often. Frequent updates of the statistics ensure that the row statistics are as up-to-date as possible.

For example, if a database contains a rolling history table that DSS queries use, run UPDATE STATISTICS on the table whenever a new set of rows is added and an old set removed.

Creating Data-Distribution Statistics

To create data distributions on specific columns in a table as well as table size statistics, use the MEDIUM or HIGH keywords with the UPDATE STATISTICS statement. If you use these keywords, the database server generates statistics about the distribution of data values for each specified column and places that information in the **sysdistrib** system catalog table. If a distribution has been generated for a column, the optimizer uses that information to estimate the number of rows that match a query against a column.

Run the UPDATE STATISTICS statement with the MEDIUM keyword for each table or for the entire database. If you run the UPDATE STATISTICS statement for the entire database, you do not need to execute a separate UPDATE STATISTICS statement for each table.

If the tables contain several tens or hundreds of columns, run an UPDATE STATISTICS statement for each table and specify only a subset of the columns. Specify columns that are most frequently used in query filters.

```
UPDATE STATISTICS MEDIUM FOR TABLE tab1(col1,col2);
UPDATE STATISTICS MEDIUM FOR TABLE tab2(col1,col3);
UPDATE STATISTICS MEDIUM FOR TABLE tab3(col2,col4);
```

You do not usually need to execute the UPDATE STATISTICS statement in HIGH mode. In MEDIUM mode, the database server samples the data to produce statistical query estimates for the columns that you specify.

Unless column values change frequently or you add rows to the table, you do not need to regenerate the data distributions. To verify the accuracy of the distribution, compare **dbschema -hd** output with the results of appropriately constructed SELECT statements. The following **dbschema** command produces a list of values for each column of table **tab2** in database **virg** and the number of rows with each specific value:

```
DBSCHEMA -hd tab2 -d virg
```

Tip: *Because UPDATE STATISTICS in MEDIUM mode uses a sampling technique to produce distribution statistics, the statistics are not always exactly the same.*

For information about using **dbschema**, refer to the *Informix Migration Guide*.

Specifying a Confidence Level for UPDATE STATISTICS

Distribution statistics have an average margin of error less than *percent* of the total number of rows in the table, where *percent* is the value that you specify in the RESOLUTION clause in MEDIUM mode. The default percent value for MEDIUM mode is 1 percent. For HIGH mode distributions, the default resolution is 0.5 percent.

If a table is large, change the default parameters for resolution and confidence:

- Specify a resolution that is smaller than the default value of `1`.
- Specify a confidence level larger than the default level of `0.95`.

  A finer resolution and a higher confidence level create a distribution closer in approximation to the HIGH mode distribution.

```
UPDATE STATISTICS MEDIUM FOR TABLE tab2(col3)
   RESOLUTION .1 .99;
```

Updating Statistics for Indexed Columns

For each table that the query accesses, use the following guidelines to build data distributions for indexed and unindexed columns:

- Run UPDATE STATISTICS MEDIUM for all columns in a table that are not the first or only column in an index. This step is a single UPDATE STATISTICS statement. The default parameters are sufficient unless the table is very large. In that case, use a resolution of `1.0`, `0.99`.

  With the DISTRIBUTIONS ONLY option, you can execute UPDATE STATISTICS MEDIUM at the table level or for the entire system because the overhead of the extra columns is not large.

- Run UPDATE STATISTICS HIGH for all columns that are indexed. For the fastest execution time of UPDATE STATISTICS, execute one UPDATE STATISTICS HIGH statement for each column.

  In addition, when you have indexes that begin with the same subset of columns, run UPDATE STATISTICS HIGH for the first column in each index that differs.

  For example, if index **ix_1** is defined on columns **a**, **b**, **c**, and **d**, and index **ix_2** is defined on columns **a**, **b**, **e**, and **f**, run UPDATE STATISTICS HIGH on column **a** by itself. Then run UPDATE STATISTICS HIGH on columns **c** and **e**. In addition, you can run UPDATE STATISTICS HIGH on column **b**, but this step is usually not necessary.

- For each multicolumn index, execute UPDATE STATISTICS LOW for *all* of its columns. For the single-column indexes in the preceding step, UPDATE STATISTICS LOW is implicitly executed when you execute UPDATE STATISTICS HIGH.

Because the statement constructs the index information statistics only once for each index, these steps ensure that UPDATE STATISTICS executes rapidly.

For additional information about data distributions and the UPDATE STATISTICS statement, see "Using Indexes" on page 13-13 and the *Informix Guide to SQL: Syntax*.

Updating Statistics for Join Columns

Because the optimizer uses statistics to choose the best query plan, you might run UPDATE STATISTICS on join columns if your query uses equality predicates:

- Run UPDATE STATISTICS statement with the HIGH keyword for specific join columns that appear in the WHERE clause of the query. If you followed the guidelines in "Updating Statistics for Indexed Columns" on page 13-11, columns that head indexes already have HIGH mode distributions.
- To determine whether HIGH mode distribution information on columns that do not head indexes can provide a better execution path, take the following steps:
  1. Issue the SET EXPLAIN ON statement and rerun the query.

     To improve performance after you run the query, issue SET EXPLAIN OFF unless you want to create SET EXPLAIN output for subsequent queries.
  2. Note the estimated number of rows in the SET EXPLAIN output and the actual number of rows that the query returns.
  3. If these two numbers are significantly different, run UPDATE STATISTICS HIGH on the columns that participate in joins, unless you have already done so.

Important: *If the table is very large, running UPDATE STATISTICS with the HIGH mode can take a long time.*

Because of improvements to cost estimates that establish better query plans, the optimizer depends greatly on an accurate understanding of the underlying data distributions in certain cases.

The following example shows a query with equi-join columns:

```
SELECT employee.name, address.city
   FROM employee, address
   WHERE employee.ssn = address.ssn
   AND employee.name = 'James'
```

In this example, the equi-join columns are the **ssn** fields in the **employee** and **address** tables. The data distributions for both of these columns must accurately reflect the actual data so that the optimizer can correctly determine the best join method and execution order.

If you are executing complex queries that involve equality filters, you might still feel that the query is not executing quickly enough with MEDIUM mode statistics. In that case, take one of the following actions:

- Run UPDATE STATISTICS statement with the HIGH keyword for specific join columns that appear in the WHERE clause of the query.
- To determine whether HIGH mode distribution information on columns that do not head indexes could provide a better execution path, take the following steps:
  1. Turn on **sqexplain** output with the SET EXPLAIN ON statement and rerun the query. Note the estimated number of rows in the **sqexplain** output.
  2. Monitor your query with either **onstat -g xqs *qryid***. Note the number of rows that the last SQL operator processed.
  3. For more information on monitoring queries, refer to "Displaying SQL Operator Statistics in the Query Plan" on page 12-26.
  4. If these two numbers are significantly different, run UPDATE STATISTICS HIGH on the columns that participate in equi-joins, unless you have already done so.

Important: *If the table is very large, UPDATE STATISTICS with the HIGH mode can take a long time to execute.*

You can use the UPDATE STATISTICS statement to create data distributions only for tables in the current database.

For additional information about data distributions and the UPDATE STATISTICS statement, see the *Informix Guide to SQL: Syntax*.

Using Indexes

You can often improve the performance of a query by creating more than one index on a table. The optimizer has more indexes to choose from in creating an efficient query plan and in some cases it can choose to use multiple indexes on a table. Analyze individual queries to help decide what indexes to create.

Important: *Although several indexes on a table might improve query performance, the more indexes that must be updated, the longer it takes to update, insert, or delete table rows. Consider the trade-off between query improvement and table-modification time for your application.*

The following sections describe how indexes improve query performance.

Preventing Repeated Sequential Scans of Large Tables

Sequential access to any table other than the first table in the plan might read every row of the table once for every row selected from the preceding tables. The number of times that tables are read determines the effect of sequential reads on performance.

If the table is small enough to reside in memory, reading it repeatedly does not degrade performance. Sequential search of an in-memory table can be faster than searching the same table through an index, especially if maintaining those index pages in memory removes other useful pages from the buffers.

If the table is larger than a few pages, however, repeated sequential access severely impairs performance.

One way to prevent this problem is to create an index on the column that is used to join the table. An index consumes disk space proportional to the width of the key values and the number of rows, as described in "Estimating Index Page Size" on page 7-6.

If active tables are used for queries, the index is updated whenever rows are inserted, deleted, or updated, which slows these operations. If space is limited on your database system or if tables are updated frequently, you can use the DROP INDEX statement to remove the index after a series of queries, which frees space and makes table updates faster.

Replacing Autoindexes with Permanent Indexes

The database server sometimes creates an *autoindex* on a large table when the optimizer determines that it is more cost effective to build the index dynamically than to scan the whole table repeatedly during execution of a nested-loop join. The **sqexplain.out** file shows when the optimizer chooses an autoindex.

If the query plan includes an *autoindex* path to a large table, you can add an index on that column to improve performance. If you rarely perform the query, you can reasonably let the database server build and discard an index. But if you perform the query often, or if you perform other queries that might use the index, create a permanent index.

Using Multiple Indexes on the Same Table

At appropriate isolation levels, the query optimizer can use more than one index to read a table in a query under the following circumstances:

- More than one index exists on the table.

  If the optimizer chooses to use a GK index, however, it cannot use any other indexes on the same table, including other GK indexes.
- Using more than one of the available indexes improves performance.
- The query or transaction is performed in Dirty Read or Committed Read isolation level, or in Repeatable Read isolation level if the entire table is locked.

The database server does not use multiple indexes at Cursor Stability isolation level. For alternative index methods, see "Using Composite Indexes" on page 13-21.

Important: *Because use of multiple indexes depends on isolation level, query plans created at one isolation level must be optimized again if they are run at a different isolation level.*

In a multiple-index scan, the database server examines only the first column in an index. This means that composite indexes can be used in a multiple-index scan. The database server does not use multiple-index scans in the following kinds of processes:

- Index aggregates, except for COUNT(*)
- Sort and group elimination
- Key-only scans
- Joins

When the Optimizer Uses Multiple Indexes on the Same Table

The database server might be able to use multiple indexes on a single table to reduce I/O to the table itself.

For example if a table **T** with columns **A** and **B** has an index **iA** on column **A** and an index **iB** on column **B**, the optimizer might choose to use both indexes for queries that require column **A** and column **B** in the WHERE clause.

When the database server uses both indexes, it scans index **iA** and generates a result set **RA** based on the WHERE clause requirements for column **A**. The database server also scans index **iB** in the same way to generate a result set **RB**. It then merges result sets **RA** and **RB** to generate result set **R** and uses result set **R** to retrieve the required data rows from table **T**. Because the merge of the result sets **RA** and **RB** into **R** eliminates many candidate rows, the database server does not need to retrieve table rows simply to evaluate them.

In some cases, however, the optimizer might not choose to use both indexes. The decision depends in part on the selectivity of the column, which the optimizer assesses from the table statistics that the UPDATE STATISTICS statement stores in the system catalog tables.

In the case of an AND operator (WHERE A = ? AND B = ?), the optimizer can consider other alternatives. A single index scan is often a better choice because it can be used for several purposes.

In the case of an OR operator (WHERE A = ? OR B = ?), the only alternative is a sequential scan. If the combination of the predicates is less than 10 percent, the optimizer chooses a multiple-index scan if the isolation level permits.

The optimizer might also choose to use multiple indexes on a single table in the following circumstances:

- For COUNT(*) queries

  If all of the predicates in the WHERE clause can be evaluated by examining indexes, the database server can count the number of matching rows in the indexes and return the result without accessing the table pages.

- For queries with NOT predicates

  To determine the list of rows that satisfy a NOT predicate, the database server can reverse the bitmap values that it obtains by evaluating indexes for a complementary predicate.

Nevertheless, if you can design a single index to meet the needs of your queries, that solution is preferable to creating multiple indexes. The single index can be used for multiple purposes during the same query execution and is less costly to maintain when the table is modified by insert, update, or delete transactions.

The following section describes an example of a multiple-index scan.

Example of a Multiple-Index Scan

The database server uses two indexes to perform the following query:

```
SELECT * FROM customer
WHERE (customer_num = "113" OR customer_num = "119")
AND order_num = "3";
```

The database server uses an index on **customer_num** to retrieve the first set of rows (`customer_num = "113" OR customer_num = "119"`) and an index on **order_num** to retrieve the other set of rows (`order_num = "3"`). Without the ability to execute a query using multiple indexes for one table, the database server would probably perform a sequential scan or use only one index.

The following excerpt from the SET EXPLAIN output shows the query path:

```
QUERY:
------
select * from customer
where(customer_num = 113 or customer_num = 119)
and order_num = 3
Estimated Cost: 1
Estimated # of Rows Returned: 1
1) informix.customer: MULTI INDEX PATH
    Filters: (informix.customer.customer_num = 113 OR
lsuto.abc.customer_num = 119 )
    (1) Index Keys: order_num (Serial, fragments: ALL)
        Index Filter: informix.customer.order_num = 3
```

To use multiple indexes to execute the query, one or more threads retrieve rows that meet the filter criteria using an index on **customer_num**, and one or more threads retrieve identifiers of rows that meet the filter criteria using an index on **order_num**. The database server converts each stream of row identifiers into a bitmap, performs an AND operation on the bitmaps to produce a final row identifier list, and uses the row identifier list to retrieve the rows from the table, as Figure 13-1 shows.

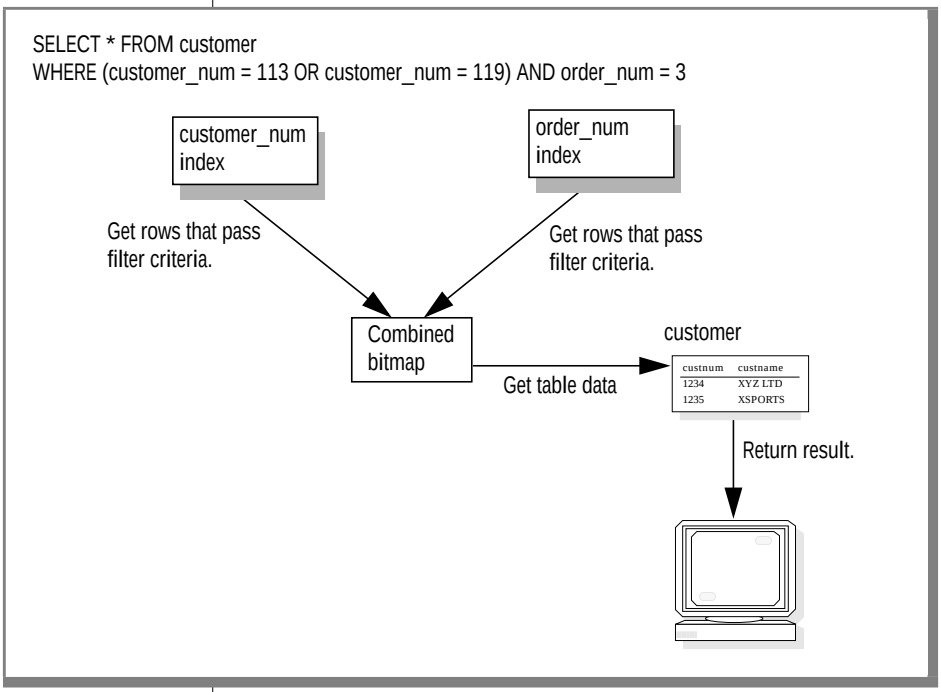


Figure 13-1
Multiple Indexes Used in a Query

Whenever possible, the database server stores the combined bitmap in the virtual portion of shared memory. If temporary storage in memory grows too large, the database server might transfer the bitmap to temporary storage on disk.

Using Key-Only Scans of Indexes

If the values contained in the index can satisfy the query, the database server does not read the table rows. This operation is called a *key-only scan*. It is faster to omit the page lookups for data pages whenever the database server can read values directly from the index. For example, if an index contains all columns required for a query that uses an aggregate function, such as COUNT or SUM, the database server can return the correct value without accessing the table.

Consider the following simple table and indexes:

```
CREATE TABLE table (col1 int, col2 int);
CREATE INDEX idx_1 on table(col1);
CREATE INDEX idx_2 on table(col2);
```

The optimizer can choose a key-only scan for the following query:

```
SELECT col1 FROM table WHERE col1 = 10;
```

However, the optimizer cannot use a key-only scan for the following query because it requires all of the values in the row:

```
SELECT * FROM table WHERE col1 = 10;
```

The database sever uses multiple-index scans as key-only scans only if the query requests a simple COUNT(*) aggregate on indexed columns. For example, the optimizer might choose to use only the **idx_1** and **idx_2** indexes to satisfy the following query:

```
SELECT COUNT(*) FROM table WHERE col1 = 10 AND col2 = 10;
```

Although the database server returns return counts of key values from multiple-index scans, it does not return the key values themselves. The optimizer cannot use a key-only scan for the following query:

```
SELECT col1, COUNT(*) FROM table WHERE col1 = 10 and col2 = 10
GROUP BY col1;
```

Even when the optimizer cannot choose a key-only index scan, however, an index lookup can improve query efficiency by skipping table pages that do not contain any required rows. When the database server uses a multiple-index scan it can also order the rows to be retrieved according to their row identifiers so that no table page needs to be read more than once.

Using Bitmap Indexes

Bitmap indexes differ from conventional B-tree indexes in the way that they store duplicate keys. A B-tree index stores a list of row identifiers for any duplicate key value. A bitmap index stores a bitmap for any highly duplicate key value. The bitmap indicates the rows that have the duplicate key value.

The more duplicate values in the key column, the more efficient the bitmap index is. For example, a bitmap index on a key column that can contain only one of two values is extremely efficient. The efficiency of the bitmap index decreases as the number of possible values increases.

To create a bitmap index, include the USING BITMAP keywords in the CREATE INDEX statement, as in the following example:

```
CREATE INDEX ix1 ON customer(status) USING BITMAP
```

For information about creating bitmap indexes and estimating their size and efficiency, refer to "Estimating Bitmap Index Size" on page 7-8.

When Bitmap Indexes Speed Queries

Bitmap indexes provide the most benefit for queries in the following circumstances:

- The key values in the index contain many duplicates.
- The index can be used in a query with the COUNT aggregate.
- The optimizer chooses to use more than one index on a table.

Consider the following example:

```
SELECT count(*) FROM customer
   WHERE zipcode = "85255" AND sales_agent_id = 22;
```

If the **customer** table has an index on **zipcode** and another index on **sales_agent_id**, the optimizer might use both indexes to execute the query. The database server creates a list of row identifiers that satisfy each part of the query and then compares the list of row identifiers to produce the final result.

Bitmap indexes excel in comparing the result of one part of the query (or predicate) to another part. The process of comparing row identifiers of rows in which **zipcode** is 85255 to row identifiers of rows in which **sales_agent_id** is 22 is faster with a bitmap index because the two lists of row identifiers are already in a form that is more efficient to compare.

Bitmap indexes also improve performance for queries that use the COUNT aggregate function. Because it is faster to count bits in a bitmap than to count row identifiers and the table need not be accessed, queries such as the following one perform better with a bitmap index:

```
SELECT COUNT(*) FROM customer
WHERE zipcode BETWEEN "85255" AND "87255" ;
```

In most other kinds of queries, bitmap indexes are comparable to B-tree indexes. Because bitmap indexes store nonduplicate key values in the same manner as B-tree indexes, performance is the same as if the index were a conventional B-tree index.

When Bitmap Indexes Decrease Index Space

If an index has many duplicate values, a bitmap index uses less disk space than a conventional B-tree index. However, the exact savings is hard to estimate because it also depends on how rows that contain the duplicate value are physically spread across the table.

Although the bitmaps are compressed, the size of the compressed bitmap depends upon the pattern of the bitmap. The practical accurate way to determine whether a bitmap index saves space is to create a bitmap index and record its size. Then drop the bitmap index and create a conventional index. Compare the size of the conventional index to the size of the bitmap index.

For more information on sizing a bitmap index, refer to "Estimating Bitmap Index Size" on page 7-8.

Using Composite Indexes

Composite indexes are commonly used to improve performance of DSS queries as well as to increase transaction efficiency for active tables in OLTP applications. A composite index can contain up to 16 keys.

You can create a composite index on a table of any type, including a temporary table.

Because a composite index indexes more than one column, it can be tailored to match the SELECT, ORDER BY, and GROUP BY clauses of the transaction or query to improve processing speed.

For example, the optimizer can use an index on the columns **a**, **b**, and **c**, in that order, in the following ways:

- For a key-only search, which is a scan that retrieves data from the index without accessing the table
- To join column **a**, columns **ab**, or columns **abc** to equivalent columns in another table
- To implement ORDER BY or GROUP BY on columns **a**, **ab**, or **abc**, but not on **b**, **c**, **ac**, or **bc**
- To locate a particular row by using equality filters followed by range expressions that use the index keys in order

  An index locates a row by specifying the first columns with equality filters and subsequent columns with range expressions (<, <=, >, >=). The following examples of filters use the columns in a composite index:

  ```
  WHERE a=1
  WHERE a>=12 AND a<15
  WHERE a=1 AND b < 5
  WHERE a=1 AND b = 17 AND c >= 40
  ```

  The following examples of filters cannot use the composite index:

  ```
  WHERE b=10
  WHERE c=221
  WHERE a>=12 AND b=15
  ```

Execution is most efficient when you create a composite index with the columns in order from most to least distinct. In other words, the first column in a composite index should be the column that returns the highest count of distinct rows when it is queried with the DISTINCT keyword of the SELECT statement.

Tip: *To see the data distribution of columns, use the* ***dbschema*** *utility, which is described in the "Informix Migration Guide."*

If your application performs several long queries, each of which contains ORDER BY or GROUP BY clauses, you can sometimes improve performance by adding indexes that produce these orderings without requiring a sort. For example, the following query sorts each column in the ORDER BY clause in a different direction:

```
SELECT * FROM t1 ORDER BY a, b DESC;
```

To avoid using temporary tables to sort column **a** in ascending order and column **b** in descending order, create a composite index on either (**a**, **b** DESC) or (**a** DESC, **b**). If your queries sort in both directions, create both indexes. For more information on bidirectional traversal of indexes, refer to the *Informix Guide to SQL: Syntax*.

On the other hand, it might be less expensive to scan the table and sort the results instead of using the composite index if the number of rows that the query retrieves does not represent a small percentage of the available data.

For large queries at appropriate isolation levels, the database server can use multiple indexes on a table instead of a composite index. Because of the order restrictions of composite indexes, multiple indexes are more flexible than composite indexes. For example, if you have a composite index on columns **a**, **b**, and **c** (in that order), an index on **b**, and an index on **c**, the optimizer can use the index on column **b** and the index on column **c** to satisfy the following query, but it cannot use the composite index:

```
SELECT * FROM tab1
WHERE b = 221 AND c = 10;
```

OLTP transactions, however, are almost always executed at an isolation level that prohibits use of multiple indexes. For this reason, designers of OLTP applications frequently create composite indexes for specific transactions.

Using Generalized-Key Indexes

Generalized-key (GK) indexes expand conventional index functionality to make indexes more useful for optimizing specific DSS queries.

Tip: *Because DSS databases often use relatively stable tables of the STATIC type, GK indexes can replace many of the uses of composite indexes. For information about how composite indexes improve performance, see* *"Using Composite Indexes" on page 13-21*.

GK indexes store the result of an expression as a key in a B-tree or bitmap index, which can be useful in specific queries on one or more large tables.

GK indexes have the following limitations:

- Only STATIC tables can have GK indexes.

  To change a table from STATIC to any other mode, you must first drop all GK indexes on the table.
- If the optimizer chooses to use a GK index on a table, it can use only one index on that table.

  If you also create other types of indexes on the table, the optimizer might choose to use more than one of these indexes, however.
- GK indexes cannot be used for key-only scans.
- GK indexes must be fragmented with the same distribution scheme as the table.

The three categories of GK indexes are as follows:

- Selective index, which contains keys for only a subset of a table
- Virtual column index, which contains keys that are the result of an expression
- Join index, which contains keys that are the result of a query that joins multiple tables

Selective Indexes

A selective index is created with a subquery that selects only a subset of rows from one or more tables.

For example, suppose you have a large table that contains orders. A common query finds the sum of all orders over a certain dollar amount, as follows:

```
SELECT sum(order_amt) FROM orders
WHERE order_amt > 1000
AND order_date > "01/01/97"
```

If you create a conventional index on **order_amt**, it contains one key (or row identifier in the case of duplicate keys) for every row. Even a large table fragmented across coservers might have an index with four or five levels. However, you can create a selective index such as the following:

```
CREATE GK INDEX ix1 ON orders
(SELECT order_amt, order_date FROM orders
   WHERE order_amt > 1000 AND order_date > "01/01/97")
```

In queries such as the preceding one, the database server can retrieve all qualified rows without reading the table first. On the other hand, with an index on only **order_amt**, the database server could use the index to retrieve all order amounts greater than `1000`, but it must still the read the table data to check that these orders have an order date greater than `01/01/97`.

In addition, if you create a selective index, the number of levels created for the index might be reduced, which would decrease the amount of disk I/O.

Virtual-Column Index

A virtual-column index contains a key that is derived from an expression. For example, suppose you commonly query on total order cost, which includes merchandise total, tax, and shipping in the following query:

```
SELECT order_num FROM orders
WHERE order_amt + shipping + tax > 1000;
```

If you add a virtual-column index to that contains the sum of **order_amt**, **shipping** and **tax**, the optimizer might choose to use the index to satisfy this query. The CREATE INDEX statement might be as follows:

```
CREATE GK index ix2 ON orders
(SELECT order_amt + shipping + tax FROM orders);
```

Without a virtual-column index in this case, the database server must scan the table sequentially to retrieve each row and add up the three columns.

Another use for a virtual-column index is for queries that search on a non-initial substring. Without a GK index, the optimizer cannot take advantage of any index on the column that contains the substring. However, the optimizer can use a GK index created on the same substring as in the query.

The following example shows how queries with noninitial substrings can use a GK index:

```
CREATE GK INDEX dept ON accts
(SELECT code[4,5] FROM accts);

SELECT * FROM accts
   WHERE code[4,5] > '50'
```

Join Indexes

A join index is created by an SQL statement that joins key columns in multiple tables to create keys in an index. The major advantage of using a join index is that the database server precomputes the join results when you create the index, eliminating the need to access some tables that are involved in a query.

You create a join index on one table, referred to here as the *indexed table*. Any other tables that are involved in the index are referred to as *foreign tables*.

Any foreign tables must be joined by their primary key or a unique key. Figure 13-2 shows a sample entity-relationship diagram, including tables for which a join index can be used effectively. The indexed table, **orders**, contains foreign keys that are joined to primary keys in two other foreign tables: **salesman** and **customer**. The relationship between the indexed table and the foreign tables is many-to-one, which guarantees that the result of the join is one row.

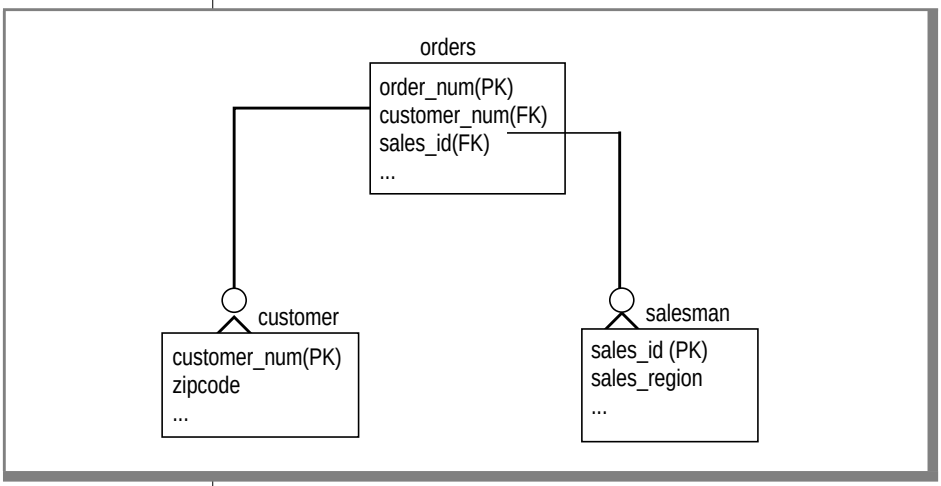


Figure 13-2
Sample Scenario for Join Indexes

Suppose that a common query retrieves all of the orders booked by salesmen in a particular region for customers with a specific zip code.

```
SELECT order_num, order_amt
FROM orders, customer, salesman
WHERE salesman.sales_region = 20
AND customer.zipcode = 89401
AND orders.customer_num = customer.customer_num
AND orders.sales_id = salesman.sales_id
```

If you create the following index, the **customer** and **salesman** tables do not need to be scanned in the previous query, resulting in a large time and resource savings:

```
CREATE GK INDEX gk_ord_sale_cust ON orders
(SELECT salesman.sales_region, customer.zipcode
FROM orders, customer, salesman
WHERE orders.customer_num = customer.customer_num
AND orders.sales_id = salesman.sales_id)
```

As the following SET EXPLAIN output shows, the query reads only the GK index on **orders** to fulfill the query:

```
QUERY:
------
select order_num, order_amt
from orders, customer, salesman
where salesman.sales_region = 20
and customer.zipcode = 89451
and orders.customer_num = customer.customer_num
and orders.sales_id = salesman.sales_id
Estimated Cost: 139
Estimated # of Rows Returned: 9
1) sales.orders: G-K INDEX PATH
   (1) Index Keys: salesman.sales_region customer.zipcode
(Parallel, fragments: ALL)
   Lower Index Filter: (lsuto.salesman.sales_region = 20 AND
sales.customer.zipcode = 89451)
```

For more information on creating a GK index with the CREATE INDEX statement, refer to the *Informix Guide to SQL: Syntax*.

Improving Filter Selectivity

The more precisely you specify the required rows, the faster your queries complete. The WHERE clause of the SELECT statement determines the amount of information that the query evaluates. The conditional expression in the WHERE clause is commonly called a *filter*.

The *selectivity* of a filter is a measure of the percentage of rows in the table that the filter can pass. The database server uses data distributions to calculate selectivities. However, in the absence of data distributions, the database server calculates default selectivities, as described in "Filter Selectivity Evaluation" on page 10-22.

To improve selectivity information for filters, schedule frequent UPDATE STATISTICS statements for the most frequently accessed tables. For more information, refer to "Creating Data-Distribution Statistics" on page 13-9.

Rewrite queries to avoid the following filter methods if possible:

- Certain difficult regular expressions
- Noninitial substrings unless the substring has a GK index

The following sections describe these types of filters and the reasons for avoiding them.

Avoiding Difficult Regular Expressions

The MATCHES and LIKE keywords support *wildcard* matches, which are technically known as *regular expressions*. Some regular expressions are more difficult than others for the database server to process. A wildcard in the initial position, as in the following example (find customers whose first names do not end in *y*), forces the database server to examine every value in the column:

```
SELECT * FROM customer WHERE fname NOT LIKE '%y'
```

Because you cannot use an index with such a filter, the database server must access the table in this example sequentially.

If a difficult test for a regular expression is essential, avoid combining it with a join. If necessary, process the single table, applying the test for a regular expression to select the required rows. Save the result in a temporary table, and join that table to the others.

Regular-expression tests with wildcards in the middle or at the end of the operand do not prevent the use of an index when one exists.

Avoiding Noninitial Substrings

A filter based on a noninitial substring of a column also requires every value in the column to be tested, as the following example shows:

```
SELECT * FROM accts
   WHERE code[4,5] > '50'
```

A standard index cannot be used to evaluate such a filter.

You might be able to create a GK index on the substring, as described in "Virtual-Column Index" on page 13-25. However, if queries must often search standard columns for noninitial substrings, the table definition might be at fault. For example, an accounting system might use a long string of numbers as hierarchical account codes. Each level of the hierarchy should be a separate column.

Improving Aggregate Statement Performance

You can improve performance for filters that contain aggregate statements if you construct appropriate indexes for the queries. Performance improvements result because of single lookups, range scans, and elimination of the GROUP SQL operator.

A filter-aggregate query has these requirements:

- It requires data from a single table.
- It uses the COUNT, MIN, or MAX keywords in the SELECT clause.
- It uses a WHERE clause that requires scanning only one index fragment to filter data.

You can improve query performance if you construct queries according to the following rules:

- The filter expression in the WHERE clause should be conjunctive, and the filters should not contain subqueries.
- The MIN or MAX aggregate should be on an index key that immediately follows major keys in the WHERE clause filters. For example, in an index with major keys A, B, and C, the next index key, D, can be the aggregate column, and A, B, and C should have exact-match filters in the WHERE clause.
- Remaining WHERE clause filters do not need to be exact-match filters and should use index keys that appear after the aggregate column in the index key list. For example, in the index that contains A, B, and C as major keys and D, E, and F as minor keys, E and F can be involved in the remaining WHERE clause filters if D is the aggregate term.

Even if the query does not contain a WHERE clause, the same performance improvement results if only one table fragment is scanned and the aggregate column is a major key of the table. For example, if table T is not fragmented and contains columns A, B, and C, with an index on columns A and B, the following query is quickly executed by scanning only the index:

```
SELECT MIN(A) from T;
```

Using SQL Extensions for Increased Efficiency

Certain SQL extensions can improve transaction and query processing if you use them appropriately. This section lists some of these SQL extensions and provides advice about their optimal use.

TRUNCATE TABLE Statement

Use the TRUNCATE TABLE statement to remove all rows from a table in a single operation and leave it ready for new content. The TRUNCATE TABLE statement is a quick way to prepare a table into which you plan to load completely new data.

When it executes the TRUNCATE TABLE statement, the database server executes a single transaction that includes a commit and writes only three records to the logical log. When the database server removes rows from the table, it frees the old extents and allocates a new extent immediately. It also removes the contents of all indexes and re-creates the indexes when the table is populated again.

To execute the TRUNCATE TABLE statement, you must be the owner of the table or have DBA privileges. You cannot use the TRUNCATE TABLE statement to remove rows from an external table or from any of the internal database tables, such as the system catalog tables or the violation tables.

The TRUNCATE TABLE statement provides the following performance advantages:

- It removes rows from a table without changing access privileges so that you do not have to re-create the table definition and assign access privileges to users.
- It does not use a time-consuming row-by-row delete operation, which also adds a record to the logical log for each deleted row, greatly increasing the size of the logical log file.

Although the database server can roll back an unsuccessful TRUNCATE TABLE operation, it cannot roll back a successful TRUNCATE TABLE operation because it is committed immediately.

DELETE USING Statement

The DELETE....USING statement lets you join two tables and delete rows from one table based on a WHERE clause that compares column values in both tables. The performance of a single statement to join tables A and B and perform a conditional delete is better than the performance of two statements, the first of which selects rows and the second of which deletes the selected rows. In addition to improving processing performance, DELETE...USING statement simplifies the syntax required to perform the operation.

For example, to delete rows in the **open** table based on a comparison with certain columns in the **bills** table, you might enter the following statement:

```
DELETE FROM open USING open, bills
   WHERE open.cust_no = bill.custno
   AND open.paid_date > bills.bill_date
   AND open.pd_amt = bills.bill_amt;
```

The only table listed in the FROM clause is the table from which rows will be deleted, but all joined tables must be listed in the USING clause. Outer joins are not allowed.

The DELETE...USING statement performs its actions in two steps. First it joins the tables and creates a temporary table based on all rows for which the WHERE clause evaluates to true. Then it deletes the matching rows in the target table.

For a complete description of the DELETE... USING statement, refer to the *Informix Guide to SQL: Syntax*.

CASE Statement

The CASE statement in SPL routines improves the performance of transactions or queries that use IF...THEN...ELSE...END IF statements.

For example, consider a transaction or query that calls an SPL routine that contains the following set of IF...THEN...ELSE...END IF statements:

```
IF i < 5 THEN                    -- 1 2 3 4
   IF i < 3 THEN                -- 1 2
      IF i < 2 THEN            -- 1
         LET i_id1, whse_id1 =
            i_id,whse_id;
      ELSE                     -- 2
         LET i_id2, whse_id2 =
            i_id, whse_id;
      END IF;
   ELSE                         -- 3 4
      IF i < 4 THEN            -- 3
         LET i_id3, whse_id3 =
            i_id, whse_id;
      ELSE                     -- 4
         LET i_id4, whse_id4 =
            i_id, whse_id;
      END IF;
   END IF;
```

You can rewrite this set of statements as a CASE statement as follows:

```
CASE (i)
   WHEN 1 THEN
      LET i_id1, whse_id1 =
         i_id,whse_id;
   WHEN 2 THEN
      LET i_id2, whse_id2 =
         i_id, whse_id;
   WHEN 3 THEN
      LET i_id3, whse_id3 =
         i_id, whse_id;
   WHEN 4 THEN
      LET i_id4, whse_id4 =
         i_id, whse_id;
   ELSE:
      RAISE EXCEPTION 100; -- Illegal value
END CASE
```

The CASE statement is easier to maintain, and it also reduces processing overhead because each condition is evaluated only once.

For detailed information about the CASE statement, refer to the *Informix Guide to SQL: Syntax*.

MIDDLE Clause

The MIDDLE clause adds selection capabilities similar to those that the FIRST clause provides. Use the MIDDLE clause to select one or more rows that fall in the middle set of values. To return the median values of an ordered set, use the ORDER BY clause.

For example, the following query selects the 10 employees with salaries in the median range:

```
SELECT MIDDLE 10 name, salary
FROM emp
ORDER BY salary DESC
```

The MIDDLE clause can replace complex SQL statements that use the FOR EACH and IF...THEN...ELIF...ENDIF construction in SPL routines or CURSOR statements to retrieve the middle rows of a sorted range, which provides a performance improvement over other formulations of this query.

Reducing the Effect of Join and Sort Operations

After you understand how the query is processed, look for ways to obtain the same output with less effort. The following suggestions can help you rewrite your query more efficiently:

- Avoid or simplify sort operations.
- Use temporary tables to reduce sorting scope.
- Use join indexes.

Avoiding or Simplifying Sort Operations

Sorting is not necessarily a liability. The sort algorithm is highly tuned and extremely efficient. It is as fast as any external sort program that you might apply to the same data. You need not avoid occasional sorts or sorts of relatively small numbers of output rows.

Avoid or reduce the scope of repeated sorts of large tables. The optimizer avoids a sort step whenever it can produce the output in its proper order automatically from an index:

- One or more of the ordered columns is not included in the index.
- The columns are named in a different sequence in the index and in the ORDER BY or GROUP BY clause.

For some queries, you can avoid large sorts by creating temporary tables, as the following section describes. If a sort is necessary, look for ways to simplify it. As discussed in "Sort-Time Costs" on page 10-26, the sort is quicker if you can sort on fewer or narrower columns.

Using Temporary Tables to Reduce Sorting Scope

Building a temporary, ordered subset of a table can speed up a query. It can help to avoid multiple-sort operations and can simplify the work of the optimizer in other ways.

For example, suppose that your application produces a series of reports on customers who have outstanding balances, one report for each major postal area, ordered by customer name. In other words, a series of queries occur, each of the following form:

```
SELECT cust.name, rcvbles.balance, ...other columns...
   FROM cust, rcvbles
   WHERE cust.customer_id = rcvbles.customer_id
     AND rcvbls.balance > 0
     AND cust.postcode LIKE '98_ _ _'
   ORDER BY cust.name
```

This query reads the entire **cust** table. For every row with the requested postal code, the database server searches the index on **rcvbles.customer_id** and performs a nonsequential disk access for every match. It writes the rows to a temporary file and sorts them. For more information on temporary files, refer to "Dbspaces for Temporary Tables and Sort Files" on page 5-10.

This procedure is acceptable if the query is performed only once, but this example includes a series of queries, each of which incurs the same amount of work.

If the customer table is fragmented by hash, an alternative is to select all customers with outstanding balances into a temporary table ordered by customer name, and to fragment the temporary table by hash so that the database server can eliminate table fragments as it processes the query. The following example shows how to create the temporary table:

```
SELECT cust.name, rcvbles.balance, ...other columns°
   FROM cust, rcvbles
   WHERE cust.customer_id = rcvbles.customer_id
      AND cvbls.balance > 0
   INTO TEMP cust_with_balance
   fragment by HASH (cust_id) in customer_dbslc;
```

You can direct queries against the temporary table in this form, as the following example shows:

```
SELECT *
   FROM cust_with_balance
   WHERE postcode LIKE '98_ _ _'
   ORDER BY cust.name
```

Each query reads the temporary table sequentially, but the table has fewer rows than the primary table.

Using Join Indexes

A join index can eliminate the need to access some tables that are involved in a query because the precomputed join results are stored in an index. For more information on join indexes, refer to "Join Indexes" on page 13-26.

Reviewing the Optimization Level

The default optimization level, HIGH, usually produces the best overall performance. The time required to optimize the query is usually insignificant. However, if testing shows that a query is still taking too long, you can set the optimization level to LOW and then check the SET EXPLAIN output to see if the optimizer chose the same query plan as before.

To specify a HIGH or LOW level of database server optimization, use the SET OPTIMIZATION statement. The *Informix Guide to SQL: Syntax* describes this statement in detail.

Reviewing the Isolation Level

The isolation level at which a query or transaction is executed affects concurrent queries and transactions. As you analyze transactions and queries, make sure that their isolation level is set appropriately.

Setting Isolation Levels for DSS Queries

DSS queries do not change table data and usually do not run against tables that are updated dynamically. For this reason, Dirty Read or Read Uncommitted are appropriate isolation levels for such queries.

Dirty Read and Read Uncommitted isolation levels do not place locks on any table or pay attention to locks that other processes set. If DSS queries are run at these isolation levels against active database tables, imprecise information might result, but the information might still be adequate for statistical analysis of large amounts of data.

Setting Isolation Levels for Transaction Processing

Transaction processing requires complete control over data integrity. For this reason, OLTP applications usually set Committed Read, Read Committed, Cursor Stability, Serializable, or Repeatable Read isolation levels.

For detailed information about the locking and concurrency effect of setting these isolation levels, see the *Informix Guide to SQL: Tutorial* and "Setting the Isolation Level" on page 8-9.

In addition, to tailor the effect of isolation levels, you can use the RETAIN UPDATE LOCKS clause of the SET ISOLATION statement. For information about how to use this clause, see "Locking and Update Cursors" on page 8-12.

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X