Performance Guide

for Informix® Dynamic Server™

Informix Dynamic Server
Informix Dynamic Server, Developer Edition
Informix Dynamic Server, Workgroup Edition

Version 7.3
February 1998
Part No. 000-4357

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Documentation Team: Diana Chase, Geeta Karmarkar, Virginia Panlasigui, Liz Suto

Table of Contents

Introduction

Read this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

This manual provides information about how to configure and operate Informix Dynamic 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 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 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 your database server is one of the following products:

- Informix Dynamic Server, Version 7.3
- Informix Dynamic Server, Developer Edition, Version 7.3.
- Informix Dynamic Server, Workgroup Edition, Version 7.3.

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 GLS (Global Language Support) locale.

This manual assumes 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 Database

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. You can use SQL scripts provided with DB-Access to derive a second database, called **sales_demo**. This database illustrates a dimensional schema for data-warehousing applications. Sample command files are also included for creating and populating these databases.

Many examples in Informix manuals are based on the **stores7** demonstration database. The **stores7** database is described in detail and its contents are listed in the *Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and the **%INFORMIXDIR%\bin** directory on Windows NT platforms. For a complete explanation of how to create and populate the **stores7** demonstration database, refer to the *DB-Access User Manual*. For an explanation of how to create and populate the **sales_demo** database, refer to the *Informix Guide to Database Design and Implementation*.

New Features

Most of the new features for Version 7.3 of Informix Dynamic Server fall into five major areas:

- Reliability, availability, and serviceability
- Performance
- Windows NT-specific features
- Application migration
- Manageability

Several additional features affect connectivity, replication, and the optical subsystem. For a comprehensive list of new features, see the release notes for the database server.

This manual includes information about the following new features:

- Enhanced query performance with the following features:
  - Enhancements to the SELECT statement to allow selection of the first *n* rows
  - Key-first index scans
  - Memory-resident tables
  - Optimizer directives
  - Enhancements to the optimization of correlated subqueries
  - Optimization goal

    Enhancements to the SET OPTIMIZATION statement

    New OPT_GOAL configuration parameter

    New **OPT_GOAL** environment variable

    Ordered merge for fragmented index scan
- Improved availability and concurrency with the following features:
  - Enhancements to the ALTER FRAGMENT statement with ATTACH and DETACH clauses to minimize index rebuilds
  - Enhancements to the ALTER TABLE statement to allow more situations to use the in-place alter algorithm
  - New **oncheck** options to increase concurrency and performance

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Command-line conventions
- Sample-code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after you type the indicated information on your keyboard. When you are instructed to "type" the text or to "press" other keys, you do not need to press* RETURN.

Icon Conventions

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

Comment Icons

Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
⚠	The *warning* icon identifies vital instructions, cautions, or critical information.
⇨	The *important* icon identifies significant information about the feature or operation that is being described.
💡	The *tip* icon identifies 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 GLS feature.
IDS	Identifies information that is specific to Dynamic Server and its editions. However, in some cases, the identified section applies only to Informix Dynamic Server and not to Informix Dynamic Server, Workgroup and Developer Editions. Such information is clearly identified.

(1 of 2)

Icon	Description
UNIX	Identifies information that is specific to the UNIX platform.
W/D	Identifies information that is specific to Informix Dynamic Server, Workgroup and Developer Editions.
WIN NT	Identifies information that is specific to the Windows NT environment.

(2 of 2)

These icons can apply to a row in a table, one or more paragraphs, or an entire 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 the feature-, product-, or platform-specific information that appears within a table or a set of paragraphs within a section.

Command-Line Conventions

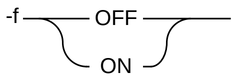
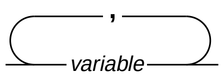
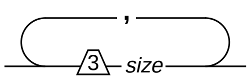
This section defines and illustrates the format of commands that are available in Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

Each diagram displays the sequences of required and optional elements that are valid in a command. A diagram begins at the upper-left corner with a command. It ends at the upper-right corner with a vertical line. Between these points, you can trace any path that does not stop or back up. Each path describes a valid form of the command. You must supply a value for words that are in italics.

You might encounter one or more of the following elements on a command-line path.

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
variable	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
.ext	A filename extension, such as **.sql** or **.cob**, might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file. The extension might be optional in certain products.
(. , ; + * - /)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
$\boxed{\text{Privileges p. 5-17}}$ $\boxed{\text{Privileges}}$	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page.
— ALL —	A shaded option is the default action.
──►──►	Syntax within a pair of arrows indicates a subdiagram.
──┤	The vertical line terminates the command.

(1 of 2)

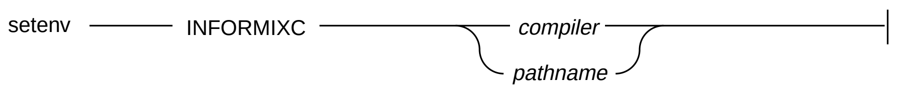
Element	Description
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items.
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify *size* no more than three times within this statement segment.

(2 of 2)

How to Read a Command-Line Diagram

Figure 1 shows a command-line diagram that uses some of the elements that are listed in the previous table.

Figure 1
Example of a Command-Line Diagram



To construct a command correctly, start at the top left with the command. Then follow the diagram to the right, including the elements that you want. The elements in the diagram are case sensitive.

Figure 1 diagrams the following steps:

1. Type the word `setenv`.
2. Type the word `INFORMIXC`.
3. Supply either a compiler name or pathname.

   After you choose *compiler* or *pathname*, you come to the terminator. Your command is complete.
4. Press RETURN to execute the command.

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 stores7
...

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 the Query-language option of 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.

Screen-Illustration Conventions

The illustrations in this manual represent a generic rendition of various windowing environments. The details of dialog boxes, controls, and windows were deleted or redesigned to provide this generic look. Therefore, the illustrations in this manual depict the **onperf** utility a little differently than the way it appears on your screen.

Additional Documentation

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

- On-line manuals
- Printed manuals
- Error message files
- 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.

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 Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. For a detailed description of these error messages, refer to *Informix Error Messages* in Answers OnLine.

UNIX

To read the error messages on UNIX, you can use the following commands.

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

♦

WIN NT

To read error messages and corrective actions on Windows NT, use the **Informix Find Error** utility. To display this utility, choose **Start→Programs→Informix** from the Task Bar. ♦

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.

UNIX

On UNIX platforms, the following on-line files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

On-Line File	Purpose
PERFDOC_7.3	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication.
SERVERS_7.3	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.
IDS_7.3	The machine-notes file describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product described.

♦

WIN NT

The following items appear in the Informix folder. To display this folder, choose **Start→Programs→Informix** from the Task Bar.

Item	Description
Documentation Notes	This item includes additions or corrections to manuals, along with information about features that might not be covered in the manuals or that have been modified since publication.
Release Notes	This item 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.

Machine notes do not apply to Windows NT platforms. ♦

Related Reading

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

For additional technical information on database management, consult the following books:

- *An Introduction to Database Systems* by C. J. Date (Addison-Wesley Publishing Company, Inc., 1995)
- *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter (Morgan Kaufmann Publishers, Inc., 1993)

To learn more about the SQL language, consult the following texts:

- *A Guide to the SQL Standard* by C. J. Date with H. Darwen (Addison-Wesley Publishing Company, Inc., 1993)
- *Understanding the New SQL: A Complete Guide* by J. Melton and A. Simon (Morgan Kaufmann Publishers, Inc., 1993)
- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

To learn more about performance measurement and tuning, consult the following texts:

- *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

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about 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 email, our address is:

doc@informix.com

Or send a facsimile to the Informix Technical Publications Department at:

650-926-6571

We appreciate your feedback.

Chapter

Performance Basics

1

This manual discusses performance measurement and tuning for Informix Dynamic 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 a general introduction to performance tuning, refer to the texts listed in the "Related Reading" on page -15.

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:
  - Balance the load on system resources
  - Adjust the configuration of your database server
  - Adjust the arrangement of your data
  - Allocate resources for decision-support queries
  - Create indexes to speed up retrieval of your data

The remainder of this chapter does the following:

- Provides a brief description of Dynamic Server parallel processing
- Describes the applications that use Dynamic Server
- Describes a basic approach for performance measurement and tuning
- Describes roles in maintaining good performance
- Lists topics that are not covered in this manual

Informix Dynamic Server

This section provides a brief description of the parallel-processing architecture and performance advantages of Dynamic Server.

Parallel-Process Architecture

Dynamic Server is a multithreaded *relational database* server that exploits symmetric multiprocessor (SMP) and uniprocessor architectures.

Scalability

Informix *dynamic scalable architecture* (DSA) describes the capability of an Informix database server to scale its resources to the demands that applications place on it.

A key element of DSA is the virtual processors that manage central processing, disk I/O, networking, and optical functions in parallel. These virtual processors take advantage of the underlying multiple processors on an SMP computer to execute SQL operations and utilities in parallel. This ability to execute tasks in parallel provides a high degree of scalability for growing workloads.

For more information on virtual processors, refer to your *Administrator's Guide*.

Client/Server Architecture

Dynamic Server is a *database server* that processes requests for data from client applications. The *client* is an application program that you run to request information from a database.

The database server accesses the requested information from its databases and sends back the results to the client applications. Accessing the database includes activities such as coordinating concurrent requests from multiple clients, performing read and write operations to the databases, and enforcing physical and logical consistency on the data.

Client applications use Structured Query Language (SQL) to send requests for data to the database server. Client programs include the DB-Access utility and programs that you write using an Informix API such as INFORMIX-ESQL/C or INFORMIX-CLI.

High Performance

Dynamic Server achieves high performance through the following mechanisms:

- Unbuffered disk management
- Dynamic shared-memory management
- Dynamic thread allocation
- Parallel Execution
- Multiple connections

The following paragraphs explain each of these mechanisms.

Unbuffered Disk Management

The database server uses unbuffered disk access to improve the speed of I/O operations.

UNIX

UNIX platforms provide unbuffered access with character-special devices (also known as *raw* disk devices). For more information about character-special devices, refer to your UNIX operating-system documentation. ♦

WIN NT

Windows NT platforms provide unbuffered access through both unbuffered files and raw disk devices. For more information about unbuffered files, refer to your Windows NT operating-system documentation. ♦

When you store tables on raw disks or in unbuffered files, the database server can manage the physical organization of data and minimize disk I/O. When you store tables in this manner, you can receive the following performance advantages:

- The database server optimizes table access by guaranteeing that rows are stored contiguously.
- The database server bypasses operating-system I/O overhead by performing direct data transfers between disk and shared memory.

For more information about how the database server uses disk space, refer to your *Administrator's Guide*.

Dynamic Shared-Memory Management

All applications that use a single instance of a database server share data in the memory space of the database server. After one application reads data from a table, other applications can access whatever data is already in memory. Through shared-memory management, the database server minimizes disk access and the associated impact on performance.

Dynamic Server shared memory contains both data from the database and control information. Because various applications need the data that is located in a single, shared portion of memory, you can put all control information needed to manage access to that data in the same place.

Dynamic Server adds memory dynamically as needed. As the administrator, you specify the size of the segment to add. For information on how to estimate the initial amount of shared memory and the amount for Dynamic Server to add dynamically, refer to "How Configuration Affects Memory Utilization" on page 3-23.

Dynamic Thread Allocation

To support multiple client applications, the database server uses a relatively small number of processes called virtual processors. A virtual processor is a multithreaded process that can serve multiple clients and, where necessary, run multiple threads to work in parallel for a single query. In this way, the database server provides a flexible architecture that provides dynamic load balancing for both on-line transaction processing (OLTP) and for decision-support applications. For complete details about dynamic scalable architecture, refer to your *Administrator's Guide*.

Parallel Execution and Fragmentation

The database server can allocate multiple threads to work in parallel on a single query. This feature is known as the parallel database query (PDQ) feature.

The database server uses local table partitioning (also called *fragmentation*) to distribute tables intelligently across disks to improve performance. If you have very large databases (VLDBs), the ability to fragment data is important if you want to efficiently manage the data.

The PDQ feature is most effective when you use it with the fragmentation feature. For information on how to use the PDQ feature, refer to Chapter 9, "Parallel Database Query." For information on how to use the fragmentation feature, refer to Chapter 6, "Fragmentation Guidelines."

Connecting Clients and Database Servers

A client application communicates with the database server through the connection facilities that the database server provides.

At the source-code level, a client connects to the database server through an SQL statement. Beyond that, the client's use of connection facilities is transparent to the application.

As the database administrator, you specify the types of connections that the database server supports in a connectivity-information file called **sqlhosts**. For more information on the connections facilities and **sqlhosts** file, refer to your *Administrator's Guide*.

You can also specify values of configuration parameters that affect the performance of these connections. For more information, refer to "NETTYPE" on page 3-18 and "Multiplexed Connections" on page 3-22.

Client Application Types

Two major classes of applications operate on data that is in a relational database:

- On-line transaction processing (OLTP) applications
- Decision-support system (DSS) applications

OLTP Applications

OLTP applications are often used to capture new data or update existing data. These operations typically involve quick, indexed access to a small number of rows. An order-entry system is a typical example of an OLTP application. OLTP applications are often multiuser applications with acceptable response times measures in fractions of seconds.

OLTP applications have the following characteristics:

- Simple transactions that involve small amounts of data
- Indexed access to data
- Many users
- Frequent requests
- Fast response times

DSS Applications

DSS applications are often used to report on or consolidate data that OLTP operations have captured over time. These applications provide information that is often used for accounting, strategic planning, and decision-making. Data within the database is typically queried but not updated during DSS operations. Typical DSS applications include payroll, inventory, and financial reports.

A Basic Approach to Performance Measurement and Tuning

Early indications of a performance problem are often vague; users might report that the system seems sluggish. Users might complain that they cannot get all their work done, that transactions take too long to complete, that queries take too long to process, or that the application slows down at certain times during the day. To determine the nature of the problem, you must measure the actual use of system resources and evaluate the results.

Users typically report performance problems in the following situations:

- Response times for transactions or queries take longer than expected.
- Transaction throughput is insufficient to complete the required workload.
- Transaction throughput decreases.

To maintain optimum performance for your database applications, develop a plan for measuring system performance, making adjustments to maintain good performance and taking corrective measures when performance degrades. Regular, specific measurements can help you to anticipate and correct performance problems. By recognizing problems early, you can prevent them from affecting your users significantly.

Informix recommends an iterative approach to optimizing Dynamic Server performance. If repeating the steps found in the following list does not produce the desired improvement, insufficient hardware resources or inefficient code in one or more client applications might be causing the problem.

To optimize performance

1. Establish your performance objectives.
2. Take regular measurements of resource utilization and database activity.
3. Identify symptoms of performance problems: disproportionate utilization of CPU, memory, or disks.
4. Tune your operating-system configuration.
5. Tune your Dynamic Server configuration.

6. Optimize your chunk and dbspace configuration, including placement of logs, sort space, and space for temporary tables and sort files.
7. Optimize your table placement, extent sizing, and fragmentation.
8. Improve your indexes.
9. Optimize your background I/O activities, including logging, checkpoints, and page cleaning.
10. Schedule backup and batch operations for off-peak hours.
11. Optimize the implementation of your database application.
12. Repeat steps 2 through 11.

Performance Goals

Many considerations go into establishing performance goals for Dynamic Server and the applications that it supports. Be clear and consistent about articulating your performance goals and priorities, so that you can provide realistic and consistent expectations about the performance objectives for your application. Consider the following questions when you establish your performance goals:

- Is your top priority to maximize transaction throughput, minimize response time for specific queries, or achieve the best overall mix?
- What sort of mix between simple transactions, extended decision-support queries, and other types of requests does the database server typically handle?
- At what point are you willing to trade transaction-processing speed for availability or the risk of loss for a particular transaction?
- Is this Dynamic 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?
- Is your configuration limited by memory, disk space, or CPU resources?

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

Measurements of Performance

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

- Throughput
- Response time
- Cost per transaction
- Resource utilization

Throughput, response time, and cost per transaction are described in the sections that follow. Resource utilization can have one of two meanings, depending on the context. The term can refer to the amount of a resource that a particular operation requires or uses, or it can refer to the current load on a particular system component. The term is used in the former sense to compare approaches for accomplishing a given task. For instance, if a given 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 term is used in the latter sense to refer, for instance, to the number of CPU cycles that are devoted to a particular query during a specific time interval.

For a discussion of the performance impacts of different load levels on various system components, refer to “Resource Utilization and Performance” on page 1-18.

Throughput

Throughput measures the overall performance of the system. For transaction processing systems, throughput is typically measured in *transactions per second* (TPS) or *transactions per minute* (TPM). Throughput depends on the following factors:

- 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

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 commit. 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 Dynamic Server logs during a given time interval. You can use the **onlog** utility to obtain a listing of logical-log records that are written to log files. You can use information from this command to track insert, delete, and update operations as well as committed transactions. However, you cannot obtain information stored in the logical-log buffer until that information is written to a log file.

If you need more immediate feedback, you can use **onstat -p** to gather an estimate. You can use the SET LOG statement to set the logging mode to unbuffered for the databases that contain tables of interest. You can also use the trusted auditing facility in Dynamic Server to record successful COMMIT WORK events or other events of interest in an audit log file. Using the auditing facility can increase the overhead involved in processing any audited event, which can reduce overall throughput. For information about the trusted auditing facility, refer to your *Trusted Facility Manual*.

Standard Throughput Benchmarks

Industry-wide 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 proud to be 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 for simple on-line transaction-processing (OLTP) comparisons. 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 with significant disk I/O activity.

- TPC-B

  This benchmark is used for stress-testing 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, which is 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.

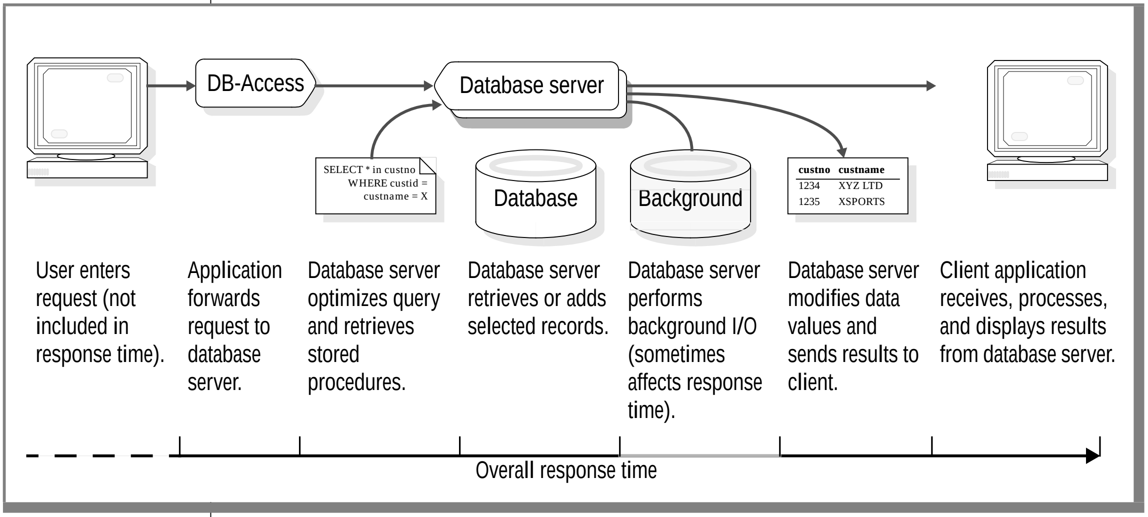
Response Time

Response time measures the performance of an individual transaction or query. Response time is typically treated as the elapsed time from the moment that a user enters a command or activates a function until the time that the application indicates the command or function has completed. The response time for a typical Dynamic Server application includes the following sequence of actions. Each action requires a certain amount of time. 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 Dynamic Server.
2. Dynamic Server performs query optimization and retrieves any stored procedures.
3. Dynamic Server retrieves, adds, or updates the appropriate records and performs disk I/O operations directly related to the query.
4. Dynamic Server performs any background I/O operations, such as logging and page cleaning, that occur during the period in which the query or transaction is still pending.
5. Dynamic Server returns a result to the application.
6. The application displays the information or issues a confirmation and then issues a new prompt to the user.

Figure 1-1 shows how these various intervals contribute to the overall response time.

Figure 1-1
Components of the Response Time for a Single Transaction


DB-Access
Database server
SELECT * in custno WHERE custid = custname = X
Database
Background
custno custname
1234 XYZ LTD
1235 XSPORTS
User enters request (not included in response time).
Application forwards request to database server.
Database server optimizes query and retrieves stored procedures.
Database server retrieves or adds selected records.
Database server performs background I/O (sometimes affects response time).
Database server modifies data values and sends results to client.
Client application receives, processes, and displays results from database server.
Overall response time

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 that the database allocates to a large query.

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

Response Time Measurement

You can use either 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

Your operating system typically has 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 that a DB-Access command file issues.

UNIX

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 timing for those commands. For more information about command files, refer to the *DB-Access User 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-defined 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 in system calls exceeds one-third 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
```

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 your 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 have a performance monitor that you can use to measure response time for a query or process.

WIN NT

You can often use the Performance Monitor that you Windows NT operating system supplies to measure the following times:

- User time
- Processor time
- Elapsed time ♦

Timing Functions Within Your 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. To measure response time, you can call **dtcurrent()** to report the time at the start of a transaction and again to report the time when the transaction commits.

Elapsed time, in a multiprogramming system or network environment where resources are shared among multiple processes, does not always correspond to execution time. Most operating systems and C libraries contain functions that return the CPU time of a program.

Cost per Transaction

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

To measure the 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 over the total number of transactions.

Although this measure is useful for planning and evaluation, it is seldom relevant to the daily issues of achieving optimum performance.

Resource Utilization and Performance

A typical transaction-processing application undergoes different demands throughout its various operating cycles. Peak loads during the day, week, month, and year, as well as the loads imposed by decision-support (DSS) queries or backup operations, can have significant impact on any system that is running near capacity. You can use direct historical data derived from your particular system to pinpoint this impact.

You must take regular measurements of the workload and performance of your system to predict peak loads and compare performance measurements at different points in your usage cycle. Regular measurements help you to develop an overall performance profile for your Dynamic Server applications. This profile is critical in determining how to improve performance reliably.

For the measurement tools that Dynamic Server provides, refer to "Capturing Database Server Performance Data" on page 2-6. For the tools that your operating system provides for measuring performance impacts on system and hardware resources, refer to "Operating-System Tools" on page 2-4.

Utilization is the percentage of time that a component is actually occupied, as compared with the total 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.

Measure and record utilization of the following system resources regularly:

- CPU
- Memory
- Disk

A resource is said to be *critical* to performance when it becomes overused or when its utilization is disproportionate to that of other components. 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 could improve performance by rearranging data to balance I/O requests across the entire set of disks.

How you measure resource utilization depends on the tools for reporting system activity and resource utilization that your operating system provides. Once you identify a resource that seems overused, you can use Dynamic 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 Dynamic Server configuration or your operating system to reduce those database activities or spread them among other components. In some cases, you might need to provide 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. Pending requests must wait for the resources to become available before they can complete. When a component is too busy to keep up with all its requests, the overused component becomes a bottleneck in the flow of activity. The higher the percentage of time that the resource is occupied, the longer each operation must wait for its turn.

You can use the following formula to estimate the service time for a request based on the overall utilization of the component that services the request. The expected service time includes the time that is spent both waiting for and using the resource in question. 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 (expressed as a decimal).

As Figure 1-2 shows, the service time for a single component increases dramatically as the utilization increases beyond 70 percent. For instance, if a transaction requires 1 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 utilization for the resource reaches 90 percent, you can expect the transaction to take 10 seconds to make its way through that component.

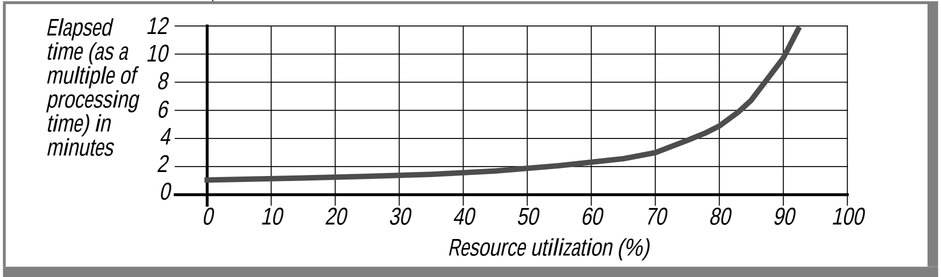


Figure 1-2
Service Time for a Single Component as a Function of Resource Utilization

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

Important: *Monitor any system resource that shows a utilization of over 70 percent or any resource that exhibits symptoms of overuse as described in the following sections.*

CPU Utilization

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

On the other hand, when 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. CPU cycles are being diverted to internal housekeeping tasks such as memory management. You can easily eliminate the following activities:

- Large queries that might be better scheduled at an off-peak time
- Unrelated application programs that might be better 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 the computer to manage. Slow response time can also indicate that the CPU is processing transactions inefficiently or that CPU cycles are being diverted.

When CPU utilization is high, a detailed analysis of the activities that Dynamic Server performs can reveal any sources of inefficiency that might be present due to improper configuration. For information about analyzing Dynamic Server activity, refer to “Capturing Database Server Performance Data” on page 2-6.

Memory Utilization

Although the principle for estimating the service time for memory is the same as that described in "Resource Utilization and Performance" on page 1-18, you use a different formula to estimate the performance impact of memory utilization than you do for other system components. Memory is not managed as a single component such as a CPU or disk, but 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.

When the operating system needs to allocate memory for use by a process, it scavenges any unused pages within memory that it can find. If no free pages exist, the memory-management system has to choose pages that other processes are still using and that seem least likely to be needed in the short run. CPU cycles are required to select those pages. The process of locating such pages is called a *page scan*. CPU utilization increases when a page scan is required.

Memory-management systems typically use a *least-recently used* algorithm to select pages that can be copied out to disk and then freed for use by other processes. When the CPU has identified pages that it can appropriate, it *pages out* the old page images by copying the old data from those pages to a dedicated disk. The disk or disk partition that stores the page images is called the *swap disk*, *swap space*, or *swap area*. This paging activity requires CPU cycles as well as I/O operations.

Eventually, page images that have been copied to the swap disk must be brought back in for use by the processes that require them. If there are still too few free pages, more must be paged out to make room. As memory comes under increasing demand and paging activity increases, this activity can reach a point at which the CPU is almost fully occupied with paging activity. A system in this condition is said to be *thrashing*. When a computer is thrashing, all useful work comes to a halt.

To prevent thrashing, some operating systems use a coarser memory-management algorithm after paging activity crosses a certain threshold. This algorithm is called *swapping*. When the memory-management system resorts to swapping, it appropriates all pages that constitute an entire process image at once, rather than a page at a time. Swapping frees up more memory with each operation. However, as swapping continues, every process that is swapped out must be read in again, dramatically increasing disk I/O to the swap device and the time required to switch between processes. Performance is then limited to the speed at which data can be transferred from the swap disk back into memory. Swapping is a symptom of a system that is severely overloaded, and throughput is impaired.

Many systems provide information about paging activity that includes the number of page scans performed, the number of pages sent out of memory (paged out), and the number of pages brought in from memory (*paged in)*:

- Paging out is the critical factor because the operating system pages out only when it cannot find pages that are free already.
- A high rate of page scans provides an early indicator that memory utilization is becoming a bottleneck.
- Pages for terminated processes are freed in place and simply reused, so paging-in activity does not provide an accurate reflection of the load on memory. A high rate of paging in can result from a high rate of process turnover with no significant performance impact.

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

$$PD \approx (C/(1-U)) * R * T$$

PD	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.

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 unacceptable.

Disk Utilization

Because each disk acts as a single resource, you can use the following 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 access time for a given disk, you can use the number of transfers per second that the operating system reports to calculate utilization for the disk. To do so, multiply the average number of transfers per second by the access time for the disk as listed by the disk manufacturer. Depending on how your data is laid out on the disk, your access times can vary from the rating of the manufacturer. 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 (this time of a disk).
A	is the access time (in seconds) that the manufacturer lists.
X	is the number of transfers per second that your operating system reports.

You can 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 (in seconds) 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 Utilization

The performance of your Dynamic Server application depends on the following factors. You must consider these factors when you attempt to identify performance problems or make adjustments to your system:

- Hardware resources

  As discussed earlier in this chapter, hardware resources include the CPU, physical memory, and disk I/O subsystems.
- Operating-system configuration

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

  The configuration of your operating system has a direct impact on how well Dynamic Server performs. The operating-system kernel takes up a significant amount of physical memory that Dynamic Server or other applications cannot use. However, you must reserve adequate kernel resources for Dynamic Server to use.
- Network configuration and traffic

  Applications that depend on a network for communication with Dynamic Server, and systems that rely on data replication to maintain high availability, are subject to the performance constraints of that network. Data transfers over a network are typically slower than data transfers from a disk. Network delays can have a significant impact on the performance of Dynamic Server and other application programs that run on the host computer.
- Dynamic Server configuration

  Characteristics of your Dynamic Server instance, such as the number of CPU virtual processors (VPs), the size of your resident and virtual shared-memory portions, and the number of users, play an important role in determining the capacity and performance of your applications.

- Dbspace, blobspace, and chunk configuration

  The following factors can affect the time that it takes Dynamic Server to perform disk I/O and process transactions:

  - The placement of the root dbspace, physical logs, logical logs, and temporary-table dbspaces
  - The presence or absence of mirroring
  - The use of devices that are buffered or unbuffered by the operation system

- Database and table placement

  The placement of tables and fragments within dbspaces, the isolation of high-use fragments in separate dbspaces, and the spreading of fragments across multiple dbspaces can affect the speed at which Dynamic Server can locate data pages and transfer them to memory.

- Tblspace organization and extent sizing

  Fragmentation strategy and the size and placement of extents can affect the ability of Dynamic Server to scan a table rapidly for data. Avoid interleaved extents and allocate extents that are sufficient to accommodate growth of a table to prevent performance problems.

- Query efficiency

  Proper query construction and cursor use can decrease the load that any one application or user imposes. Remind users and application developers that others require access to the database and that each person's activities affect the resources that are available to others.

- Scheduling background I/O activities

  Logging, checkpoints, page cleaning, and other operations, such as making backups or running large decision-support queries, can impose constant overhead and large temporary loads on the system. Schedule backup and batch operations for off-peak times whenever possible.

- Remote client/server operations and distributed join operations

  These operations have an important impact on performance, especially on a host system that coordinates distributed joins.
- Application-code efficiency

  Application programs introduce their own load on the operating system, the network, and Dynamic Server. These programs can introduce performance problems if they make poor use of system resources, generate undue network traffic, or create unnecessary contention in Dynamic Server. Application developers must make proper use of cursors and locking levels to ensure good Dynamic 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.
- the placement of tables and fragments.
- how the distribution of data across disks affects performance.

Application developers should:

- carefully design applications 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 your "Administrator's Guide" and your "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 capacity that is inadequate 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

Performance Monitoring

2

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 to create a performance history, to monitor performance at scheduled times, or to monitor ongoing system performance.

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 OLTP (online transaction processing) systems are different from the causes of problems on systems that are used primarily for DSS query applications. Systems with mixed use provide a greater performance-tuning challenge and require a sophisticated analysis of performance-problem causes.

Creating a Performance History

As soon as you set up your database server and begin to run applications on it, you should begin scheduled monitoring of resource use. To accumulate data for performance analysis, use the command-line utilities described in "Capturing Database Server Performance Data" on page 2-6 and "Operating-System Tools" on page 2-4 in operating scripts or batch files.

The Importance of a Performance History

To build up a performance history and profile of your system, take regular snapshots of resource-utilization information. For example, if you chart the CPU utilization, paging-out rate, and the I/O transfer rates for the various disks on your system, you can begin to identify peak-use levels, peak-use intervals, and heavily loaded resources. If you monitor fragment use, you can determine whether your fragmentation scheme is correctly configured. Monitor other resource use as appropriate for your database server configuration and the applications that run on it.

If you have this 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 command-line utilities that you can run at regular intervals from scripts or batch files. You also use performance monitoring tools with a graphical interface to keep an eye on 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 displays.

UNIX

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

UNIX Utility	Description
vmstat	Displays virtual-memory statistics.
iostat	Displays I/O utilization statistics.
sar	Displays a variety of resource statistics.
ps	Displays active process information.

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. ♦

WIN NT

Your Windows NT operating-system supplies a Performance Monitor (**perfmon.exe**) that can monitor resources such as processor, memory, cache, threads, and processes. The Performance Monitor also provides charts, alerts, report capabilities, and the ability to save information to log files for later analysis.

For more information on how to use the Performance Monitor, consult your operating-system manuals. ♦

Capturing Database Server Performance Data

The database server provides utilities to capture snapshot information about your configuration and performance. It also provides the system-monitoring interface (SMI) for monitoring performance from within your application.

You can use these utilities regularly to build a historical profile of database activity, which you can compare with operating-system resource-utilization data. These comparisons can help you discover which Dynamic Server activities have the greatest impact on system-resource utilization. You can use this information to identify and manage your high-impact activities or adjust your Dynamic Server or operating-system configuration.

Dynamic Server provides the following utilities:

- **onstat**
- **onlog**
- **oncheck**
- ON-Monitor
- DB-Access and the system-monitoring interface (SMI)
- **onperf**
- DB/Cockpit

You can use **onstat**, **onlog**, or **oncheck** commands invoked by the **cron** scheduling facility to capture performance-related information at regular intervals and build a historical performance profile of your Dynamic Server application. The following sections describe these utilities.

You can use ON-Monitor to check the current Dynamic Server configuration. For information about ON-Monitor, refer to your *Administrator's Guide*.

You can use **cron** and SQL scripts with DB-Access to query SMI tables at regular intervals. For information about SQL scripts, refer to the *DB-Access User Manual*. For information about SMI tables, refer to your *Administrator's Guide*.

You can use **onperf** to display Dynamic Server activity with the Motif window manager. For information about **onperf**, refer to Chapter 11, "The onperf Utility on UNIX."

DB/Cockpit is another graphical utility that you can use to view or adjust Dynamic Server activity. For information about DB/Cockpit, refer to the *DB/Cockpit User Manual*.

The onstat Utility

You can use the **onstat** utility to check the current status of Dynamic Server and monitor the activities of Dynamic Server. This utility displays a wide variety of performance-related and status information. For a complete list of all **onstat** options, use **onstat - -**. For a complete display of all the information that **onstat** gathers, use **onstat -a**.

The following table lists **onstat** options that display performance-related information.

Option	Performance Information
onstat -p	Performance profile
onstat -b	Buffers currently in use
onstat -l	Logging information
onstat -x	Transactions
onstat -u	User threads
onstat -R	Least-recently used (LRU) queues
onstat -F	Page-cleaning statistics
onstat -g	General information

The **onstat -g** option accepts a number of arguments to specify further the information to be displayed. The following list includes only those arguments that might typically serve as starting points when you track performance problems. For a list and explanation of **onstat -g** arguments, refer to your *Administrator's Guide*.

The following arguments to **onstat -g** pertain to CPU utilization.

Argument	Description
act	Displays active threads.
ath	Displays all threads. The **sqlexec** threads represent portions of client sessions; the **rstcb** value corresponds to the **user** field of the **onstat -u** command.
glo	Displays global multithreading information, including CPU-use information about virtual processors, the total number of sessions, and other multithreading global counters.
ntd	Displays network statistics by service.
ntt	Displays network user times.
ntu	Displays network user statistics.
qst	Displays queue statistics.
rea	Displays ready threads.
sch	Displays the number of semaphore operations, spins, and busy waits for each VP.
ses *session id*	Displays session information by *session id*. If you omit *session id*, this option displays one-line summaries of all active sessions.
sle	Displays all sleeping threads.
spi	Displays longspins, which are spin locks that virtual processors have spun more than 10,000 times in order to acquire. To reduce longspins, reduce the number of virtual processors, reduce the load on the computer or, on some platforms, use the *no-age* or *processor affinity* features.
sql *session id*	Displays SQL information by session. If you omit *session id*, this argument displays summaries of all sessions.
sts	Displays maximum and current stack use per thread.

(1 of 2)

Argument	Description
tpf *tid*	Displays a thread profile for *tid*. If *tid* is 0, this argument displays profiles for all threads.
wai	Displays waiting threads, including all threads waiting on mutex or condition, or yielding.
wst	Displays wait statistics.

(2 of 2)

The following arguments to **onstat -g** pertain to memory utilization.

Argument	Description
ffr *pool name | session id*	Displays free fragments for a pool of shared memory or by session.
dic *table*	Displays one line of information for each table cached in the shared-memory dictionary. If you provide a specific table name as a parameter, this argument displays internal SQL information about that table.
iob	Displays big-buffer use by I/O virtual processor class.
mem *pool name | session id*	Displays memory statistics for the pools that are associated with a session. If you omit *pool_name | session id*, this argument displays pool information for all sessions.
mgm	Displays memory grant manager resource information.
nsc *client id*	Displays shared-memory status by client ID. If you omit *client id*, this argument displays all client status areas.
nsd	Displays network shared-memory data for poll threads.
nss *session id*	Displays network shared memory status by *session id*. If you omit *session id*, this argument displays all session status areas.
seg	Displays shared-memory-segment statistics. This argument shows the number and size of all attached segments.
ufr *pool name | session id*	Displays allocated pool fragments by user or session.

The following arguments to **onstat -g** pertain to disk utilization.

Argument	Description
iof	Displays asynchronous I/O statistics by chunk or file. This argument is similar to the **onstat -d**, except that information on nonchunk files also appears. This argument displays information about temporary dbspaces and sort files.
iog	Displays asynchronous I/O global information.
ioq	Displays asynchronous I/O queuing statistics.
iov	Displays asynchronous I/O statistics by virtual processor.

For a detailed case study that uses various **onstat** outputs, refer to "Case Studies and Examples" on page A-1.

The onlog Utility

The **onlog** utility displays all or selected portions of the logical log. This command can take input from selected log files, the entire logical log, or a backup tape of previous log files. The **onlog** utility can help you to identify a problematic transaction or to gauge transaction activity that corresponds to a period of high utilization, as indicated by your periodic snapshots of database activity and system-resource consumption.

Use **onlog** with caution when you read logical-log files still on disk because attempting to read unreleased log files stops other database activity. For greatest safety, Informix recommends that you wait to read the contents of the logical-log files until after you have backed up the files and then read the files from tape. With proper care, you can use the **onlog -n** option to restrict **onlog** only to logical-log files that have been released. To check on the status of logical-log files, use **onstat -l**. For more information about **onlog**, refer to your *Administrator's Guide*.

The oncheck Utility

The **oncheck** utility displays information about storage structures on a disk, including chunks, dbspaces, blobspaces, extents, data rows, system catalog tables, and other options. You can also use **oncheck** to rebuild an index that resides in the same dbspace as the table that it references.

The **oncheck** utility provides the following options and information.

Option	Information
-pB	Blobspace blob (TEXT or BYTE data)
-pc	Database system catalog tables
-pd	Data-row information (without TEXT or BYTE data)
-pD	Data-row information (with TEXT or BYTE data)
-pe	Chunks and extents
-pk	Index key values
-pK	Index keys and row IDs
-pl	Index leaf key values
-pL	Index leaf key values and row IDs
-pp	Pages by table or fragment
-pP	Pages by chunk
-pr	Root reserved pages
-pt	Space used by table or fragment
-pT	Space used by table, including indexes

For more information about using **oncheck** to monitor space, refer "Estimating Table and Index Size" on page 4-8. For more information on concurrency during **oncheck** execution, refer "Checking Indexes" on page 4-25. For more **oncheck** information, refer to your *Administrator's Guide*.

Chapter

3

Configuration Impacts on Performance

This chapter describes the performance impacts of various operating-system and configuration parameters. This information can help you configure Dynamic Server for improved CPU, memory, and disk utilization. This chapter assumes that your database server is running and that you have followed the configuration guidelines described in your *Administrator's Guide*.

Your Current Configuration

Before you begin to adjust the configuration of your database serve, evaluate the performance of your current configuration. When you alter certain Dynamic Server characteristics, you must bring down the database server, which can affect your production system. Some configuration adjustments can unintentionally decrease performance or cause other negative side effects.

If your database applications perform well enough to satisfy user expectations, you should avoid frequent adjustments, even if those adjustments might theoretically improve performance. As long as your users are reasonably satisfied, take a measured approach when you reconfigure Dynamic Server. When possible, use a test instance of Dynamic Server to evaluate configuration changes before you reconfigure your production system.

To examine your current Dynamic Server configuration, you can use the utilities described in Chapter 2, “Performance Monitoring.”

When performance problems relate to backup operations, you might also examine the number or transfer rates for tape drives. You might need to alter the layout or fragmentation of your tables to reduce the impact of backup operations. For information about disk layout and table fragmentation, refer to Chapter 4, “Table and Index Performance Considerations.”

For client / server configurations, consider network performance and availability. Evaluating network performance is beyond the scope of this manual. For information on monitoring network activity and improving network availability, see your network administrator or refer to the documentation for your networking package.

How Configuration Affects CPU Utilization

This section discusses how the combination of operating-system and Dynamic Server configuration parameters can affect CPU utilization. This section discusses the parameters that most directly affect CPU utilization and how to set them. When possible, this section also describes considerations and recommends parameter settings that might apply to different types of workloads.

Multiple Dynamic Server database servers that run on the same host computer perform poorly when compared with a single Dynamic Server instance that manages multiple databases. Multiple database servers cannot balance their loads as effectively as a single database server. Avoid multiple residency for production environments in which performance is critical.

UNIX

UNIX Parameters That Affect CPU Utilization

Your Dynamic Server distribution includes a computer-specific file called **$INFORMIXDIR/release/IDS_7.3**, which contains recommended values for UNIX configuration parameters. Compare the values in this file with your current operating-system configuration.

The following UNIX parameters affect CPU utilization:

- 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. Semaphores for Dynamic Server are in addition to any that you allocate for other software packages.

Each instance of Dynamic Server requires the following semaphore sets:

- One set for each group of up to 100 virtual processors (VPs) that are initialized with Dynamic Server
- One set for each additional VP that you might add dynamically while Dynamic Server is running
- One set for each group of 100 or fewer user sessions connected through the shared-memory communication interface

Tip: *For best performance, Informix recommends that you allocate enough semaphores for double the number of* `ipcshm` *connections that you expect. Informix also recommends that you use the NETTYPE parameter to configure Dynamic Server poll threads for this doubled number of connections. For a description of poll threads, refer to your "Administrator's Guide." For information on configuring poll threads, refer to "NETTYPE" on page 3-18.*

Because utilities such as **onmode** use shared-memory connections, you must configure a minimum of two semaphore sets for each instance of Dynamic Server: one for the initial set of VPs and one for the shared-memory connections that Dynamic Server utilities use. The SEMMNI operating-system configuration parameter typically specifies the number of semaphore sets to allocate. For information on how to set semaphore-related parameters, refer to the configuration instructions for your operating system.

The SEMMSL operating-system configuration parameter typically 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, typically given by the SEMMNS operating-system configuration parameter. Use the following formula to calculate the total number of semaphores that each instance of Dynamic Server requires:

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

init_vps	is the number of VPs that are initialized with Dynamic Server. This number includes CPU, PIO, LIO, AIO, SHM, TLI, SOC, and ADM VPs. (For a description of these VPs, see your *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 Dynamic Server.
concurrent_utils	is the number of concurrent Dynamic Server utilities that can connect to this instance. Informix suggests that you allow for a minimum of six utility connections: two for ON-Archive and four for other utilities such as **onstat**, ON-Monitor, and **oncheck**.

If you use software packages that require semaphores in addition to those that Dynamic Server needs, the SEMMNI configuration parameter must include the total number of semaphore sets that Dynamic Server and your other software packages require. You must set the SEMMSL configuration parameter to 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. To specify this limit, you can use an operating-system configuration parameter, typically NOFILE, NOFILES, NFILE, or NFILES. The number of open-file descriptors that each instance of Dynamic Server needs is determined by the number of chunks in your database, the number of VPs that you run, and the number of network connections that your Dynamic Server instance must support. Network connections include all but those specified as the `ipcshm` connection type in either the **sqlhosts** file or a NETTYPE Dynamic Server configuration entry. Use the following formula to calculate the number of file descriptors that your instance of Dynamic Server requires:

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

chunks	is the number of chunks that are to be configured.
net_connections	is the number of network connections (those other than `ipcshm`) that your instance of Dynamic 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 impact other resources, including CPU and I/O. Insufficient physical memory for the overall system load can lead to thrashing, as section “Memory Utilization” on page 1-22 describes. Insufficient memory for Dynamic Server can result in excessive buffer-management activity. For more information on configuring memory, refer to “Configuring UNIX Shared Memory” on page 3-29.

Configuration Parameters and Environment Variables That Affect CPU Utilization

The following parameters in the Dynamic Server configuration file have a significant impact on CPU utilization:

- NUMCPUVPS
- SINGLE_CPU_VP
- MULTIPROCESSOR
- NOAGE
- AFF_NPROCS
- AFF_SPROC
- NUMAIOVPS
- OPTCOMPIND
- MAX_PDQPRIORITY
- DS_MAX_QUERIES
- DS_MAX_SCANS
- NETTYPE

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

The following environment variables affect CPU utilization:

- **PSORT_NPROCS**
- **PDQPRIORITY**
- **OPTCOMPIND**

PSORT_NPROCS, when set in the environment of a client application, indicates the number of parallel sort threads that the application can use. Dynamic Server imposes an upper limit of 10 sort threads per query for any application. For more information on parallel sorts and **PSORT_NPROCS**, see "Parameters and Variables That Affect Temporary Tables and Sorting" on page 3-51.

The **PDQPRIORITY** environment variable, when set in the environment of a client application, places a limit on the percentage of CPU VP utilization, shared memory, and other resources that can be allocated to any query that the client starts.

A client can also use the SET PDQPRIORITY statement in SQL to set a value for PDQ priority. The actual percentage allocated to any query is subject to the factor that the MAX_PDQPRIORITY configuration parameter sets. For more information on how to limit resources that can be allocated to a query, see "MAX_PDQPRIORITY" on page 3-15.

The **OPTCOMPIND** environment variable, when set in the environment of a client application, indicates the preferred way to perform join operations. This variable overrides the value that the OPTCOMPIND configuration parameter sets. For details on how to select a preferred join method, refer to "OPTCOMPIND" on page 3-15.

For more information about environment variables that affect Informix database servers, refer to the *Informix Guide to SQL: Reference*.

NUMCPUVPS, MULTIPROCESSOR, and SINGLE_CPU_VP

The NUMCPUVPS parameter specifies the number of CPU VPs that Dynamic Server brings up initially. Do not allocate more CPU VPs than there are CPUs available to service them. For uniprocessor computers, Informix recommends that you use one CPU VP. For multiprocessor systems with four or more CPUs that are primarily used as database servers, Informix recommends that you set NUMCPUVPS to one less than the total number of processors. For multiprocessor systems that you do not use primarily 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. To test if performance improves, set NUMCPUVPS to 1 in your ONCONFIG file and then add a CPU VP dynamically at runtime with **onmode -p**.

If you are running multiple CPU VPs, set the MULTIPROCESSOR parameter to 1. When you set MULTIPROCESSOR to 1, Dynamic Server performs locking in a manner that is appropriate for a multiprocessor. Otherwise, set this parameter to 0.

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 the latter is greater than 1, Dynamic 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 Dynamic Server is in on-line mode.

The number of CPU VPs is used as a 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. 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 **onstat -g ath** to monitor the number of scan threads per CPU VP or use **onstat -g ses** to focus on a particular session.

NOAGE

The NOAGE parameter allows you to disable process priority aging for Dynamic Server CPU VPs on operating systems that support this feature. (For information on whether your version of Dynamic Server supports this feature, refer to the machine-notes file.) Set NOAGE to 1 if your operating system supports this feature.

AFF_NPROCS and AFF_SPROC

Dynamic Server supports automatic binding of CPU VPs to processors on multiprocessor host computers that support processor affinity. You can use processor affinity to distribute the computation impact of CPU VPs and other processes. On computers that are dedicated to Dynamic Server, assigning CPU VPs to all but one of the CPUs achieves maximum CPU utilization. On computers that support both Dynamic Server and client applications, you can bind applications to certain CPUs through the operating system. By doing so, you effectively reserve the remaining CPUs for use by Dynamic Server CPU VPs, which you bind to the remaining CPUs using the AFF_NPROCS and AFF_SPROC configuration parameters.

On a system that runs Dynamic Server and client (or other) applications, you can 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 require waiting for user input. Because AIO VP activity usually comes in quick bursts followed by idle periods waiting for the disk, you can often interleave client and I/O operations without their unduly impacting each other.

Binding a CPU VP to a processor does not prevent other processes from running on that processor. Application (or other) processes that you do not bind to a CPU are free to run on any available processor. On a computer that is dedicated to Dynamic 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 once it arrives from disk.

The AFF_NPROCS and AFF_SPROC parameters specify the number of CPU VPs to bind using processor affinity and the processors to which those VPs are bound. (For information on whether your version of Dynamic Server supports processor affinity, refer to the machine-notes file.) When you assign a CPU VP to a specific CPU, the VP runs only on that CPU; 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 3-11). 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. You typically 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. Dynamic Server assigns CPU VPs to CPUs serially, starting with that CPU. To avoid a certain CPU, set AFF_SPROC to 1 plus the number of that CPU.

NUMAIOVPS

The NUMAIOVPS parameter indicates the number of AIO VPs that Dynamic Server brings up initially. If your operating system does not support kernel asynchronous I/O (KAIO), Dynamic 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. You can add an additional AIO VP for each chunk that Dynamic Server accesses frequently.

The machine-notes file for your version of Dynamic Server indicates whether the operating system supports KAIO. If KAIO is supported, the machine-notes describe how to enable KAIO on your specific operating system.

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

The goal in allocating AIO VPs is to allocate enough of them so that the lengths of the I/O request queues are kept short (that is, the queues 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. The **onstat -g ioq** command allows you to monitor the length of the I/O queues for the AIO VPs.

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

OPTCOMPIND

The OPTCOMPIND parameter helps the optimizer choose an appropriate access method for your application. When the optimizer examines join plans, OPTCOMPIND indicates the preferred method for performing the join operation for an ordered pair of tables. If OPTCOMPIND is equal to 0, the optimizer gives preference to an existing index (nested-loop join) even when a table scan might be faster. If OPTCOMPIND is set to 1 and the isolation level for a given query is set to Repeatable Read, the optimizer uses nested-loop joins. When OPTCOMPIND is equal to 2 (its default value), the optimizer selects a join method based on cost alone even though table scans can temporarily lock an entire table. For more information on OPTCOMPIND and the different join methods, see "How OPTCOMPIND Affects the Query Plan" on page 7-7.

To set the value for OPTCOMPIND for specific applications or user sessions, set the **OPTCOMPIND** environment variable for those sessions. Values for this environment variable have the same range and semantics as they have for the configuration parameter.

MAX_PDQPRIORITY

The MAX_PDQPRIORITY parameter limits the percentage of parallel database queries (PDQ) resources that a query can occupy. Use this parameter to limit the impact of large CPU-intensive queries on transaction throughput.

You specify this parameter as an integer that represents a percentage of the following PDQ resources that a query can request:

- Memory
- CPU VPs
- Disk I/O
- Scan threads

When a query requests a percentage of PDQ resources, Dynamic Server allocates the MAX_PDQPRIORITY percentage of the amount requested, as the following formula shows:

```
Resources allocated = PDQPRIORITY/100 * MAX_PDQPRIORITY/100
```

For example, if a client uses the SET PDQPRIORITY 80 statement to request 80 percent of PDQ resources, but MAX_PDQPRIORITY is set to 50, the database server allocates only 40 percent of the resources (50 percent of the request) to the client.

Viewed from the perspective of decision support and on-line transaction processing (OLTP), setting MAX_PDQPRIORITY allows the Dynamic Server administrator to control the impact that individual decision-support queries have on concurrent OLTP performance. Reduce the value of MAX_PDQPRIORITY when you want to allocate more resources to OLTP processing. Increase the value of MAX_PDQPRIORITY when you want to allocate more resources to decision-support processing.

For more information on how to control the use of PDQ resources, refer to "Allocating Resources for PDQ Queries" on page 9-7.

DS_MAX_QUERIES

The DS_MAX_QUERIES parameter specifies a maximum number of decision-support queries that can run at any one time. In other words, DS_MAX_QUERIES controls only queries whose PDQ priority is nonzero. Queries with a low PDQ priority value consume proportionally fewer resources, so a larger number of those queries can run simultaneously. You can use the DS_MAX_QUERIES parameter to limit the performance impact of CPU-intensive queries.

Dynamic Server uses the value of DS_MAX_QUERIES with DS_TOTAL_MEMORY to calculate quantum units of memory to allocate to a query. For more information on how Dynamic Server allocates memory to queries, refer to "DS_TOTAL_MEMORY" on page 3-38.

DS_MAX_SCANS

The DS_MAX_SCANS parameter limits the number of PDQ scan threads that can run concurrently. This parameter prevents Dynamic Server from being flooded with scan threads from multiple decision-support queries.

To calculate the number of scan threads allocated to a query, use the following formula:

```
scan_threads  = min (nfrags, (DS_MAX_SCANS * pdqpriority / 100
                * MAX_PDQPRIORITY / 100) )
```

nfrags	is the number of fragments in the table with the largest number of fragments.
pdqpriority	is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

Reducing the number of scan threads can reduce the time that a large query waits in the ready queue, particularly when many large queries are submitted concurrently. However, if the number of scan threads is less than `nfrags`, the query takes longer once it is underway. For example, if a query needs to scan 20 fragments in a table, but the `scan_threads` formula lets the query begin when only 10 scan threads are available, each scan thread scans two fragments serially, making query execution approximately twice as long as it would be if 20 scan threads were used.

NETTYPE

The NETTYPE parameter configures poll threads for each connection type that your instance of Dynamic Server supports. If your Dynamic Server instance supports connections over more than one interface or protocol, you must specify a separate NETTYPE parameter for each connection type.

You typically include a separate NETTYPE parameter for each connection type that is associated with a dbservername. Dbservernames are listed in the DBSERVERNAME and DBSERVERALIASES configuration parameters. Connection types are associated with dbservernames through entries in the **sqlhosts** file. For information about connection types and the **sqlhosts** file, refer to your *Administrator's Guide*. The first NETTYPE entry for a given connection type applies to all dbservernames associated with that type. Subsequent NETTYPE entries for that connection type are ignored. NETTYPE entries are required for connection types that are used for outgoing communication only, even if those connection types are not listed in the **sqlhosts** file.

Specifying Virtual Processor Classes for Poll Threads

Each poll thread configured or added dynamically by a NETTYPE entry runs in a separate VP. There are two VP classes in which a poll thread can run: NET and CPU. For best performance, Informix recommends that you use a NETTYPE entry to assign only one poll thread to the CPU VP class 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.

Specifying Number of Connections and Number of Poll Threads

The optimum number of connections per poll thread is approximately 300 for uniprocessor computers and 350 for multiprocessor computers. However, a poll thread can support 1,024 or perhaps more 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, using the following comma-separated fields. There can be no white space 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 typically set this field to match the *connection_type* field of a dbservername entry that is 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.

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 virtual processor 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 dbservername 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 dbservername that is given in the DBSERVERALIASES parameter, 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 3-6.*

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

If your computer is a uniprocessor and your Dynamic 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 Dynamic 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*.

Threshold for Free Buffers in Network Buffer Pool

Dynamic Server implements a new threshold of free network buffers to prevent frequent allocations and deallocations of shared memory for the network bufferpool. This new threshold enables the database server to correlate the number of free network buffers with the number of connections that you specify in the NETTYPE configuration parameter.

The database server dynamically allocates network buffers for request messages from clients. After the database server processes client requests, it returns buffers to the network free bufferpool.

If the number of free buffers is greater than the threshold, the database server returns the memory allocated to buffers over the threshold to the global pool.

The database server uses the following formula to calculate the threshold for the free buffers in the network buffer pool:

```
free network buffers threshold =
                     100 + (0.7 * number_connections)
```

The value for `number_connections` is the total number of connections that you specified in the third field of the NETTYPE entry for the different type of network connections (SOCTCP, IPCSTR, or TLITCP). This formula does not use the NETTYPE entry for shared memory (IPCSHM).

If you do not specify a value in the third field of the NETTYPE parameter, the database server uses the default value of 50 connections.

Virtual Processors and CPU Utilization

While Dynamic Server is on-line, it allows you to start and stop VPs that belong to certain classes. You can use **onmode -p** or ON-Monitor to start additional VPs for the following classes while Dynamic Server is on-line: CPU, AIO, PIO, LIO, SHM, TLI, and SOC. You can drop VPs of the CPU class only while Dynamic Server is on-line.

Whenever you add a network VP (SOC or TLI), 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 indicates that an available CPU VP can handle the poll thread, Dynamic Server assigns the poll thread to that CPU VP. If all the CPU VPs have poll threads assigned to them, Dynamic Server adds a second network VP to handle the poll thread.

Multiplexed Connections

Many traditional nonthreaded SQL client applications use multiple database connections to perform work for a single user. Each database connection establishes a separate network connection to the database server.

The Dynamic Server multiplexed connection facility provides the ability for one network connection in the database server to handle multiple database connections from an ESQL/C application to this database server.

When a nonthreaded client uses a multiplexed connection, the database server still creates the same number of user sessions and user threads as with a nonmultiplexed connection. However, the number of network connections decreases when a you use multiplexed connections. Instead, the database server uses a multiplex listener thread to allow the multiple database connections to share the same network connection.

Nonthreaded clients can experience an improved response time when you use multiplexed connections to execute SQL queries. The amount of performance improvement depends on the following factors:

- The decrease in total number of network connections and the resulting decrease in system CPU time

  The usual cause for large system CPU time is the processing of system calls for the network connection. Therefore, the maximum decrease in system CPU time is proportional to the decrease in the total number of network connections.

- The ratio of this decrease in system CPU time to the user CPU time

  If the queries are simple and use little user CPU time, you might experience a sizable reduction in response time when you use a multiplexed connection. But if the queries are complex and use a large amount of user CPU time, you might not experience a performance improvement.

  To get an idea of the amounts of system CPU time and user CPU times per virtual processor, use the **onstat -g glo** option.

To use multiplexed connections for a nonthreaded client application, you must take the following steps before you bring up the database server:

1. Add a NETTYPE entry to your ONCONFIG file with `SQLMUX` specified in the **connection_type** field.

   The NETTYPE SQLMUX configuration parameter tells the database server to create the multiplex listener thread. When you specify `SQLMUX` in the **connection_type** field of the NETTYPE configuration parameter, the other NETTYPE fields are ignored.

2. Set the multiplexed option (`m=1`) in the client **sqlhosts** file for the corresponding dbservername entry.

   For more details on the ONCONFIG file NETTYPE entry and the **sqlhosts** entry, refer to your *Administrator's Guide*.

   For more information on restrictions on the use of multiplexed connections, refer to the *INFORMIX-ESQL/C Programmer's Manual* and your *Administrator's Guide*.

How Configuration Affects Memory Utilization

This section discusses how the combination of operating system and Dynamic Server configuration parameters can affect memory utilization. This section discusses the parameters that most directly affect memory utilization and explains how to set them. Where possible, this section also provides suggested settings or considerations that might apply to different workloads.

You must consider the amount of physical memory that is available on your host when you allocate shared memory for Dynamic Server. In general, if you increase space for Dynamic Server shared memory, you can enhance the performance of your database server. You must balance the amount of shared memory dedicated to Dynamic Server against the memory requirements for VPs and other processes.

Allocating Shared Memory

You must configure adequate shared-memory resources for Dynamic Server in your operating system. 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 Dynamic Server attaches all or part of a shared-memory segment, it is called a *portion*.

Dynamic Server uses the following shared-memory portions. Each portion makes a separate contribution to the total amount of shared memory that Dynamic Server requires:

- The resident portion
- The virtual portion
- The message portion

The resident and message portions are static; you must allocate sufficient memory for them before you bring Dynamic Server into on-line mode. (Typically, you must reboot the operating system to reconfigure shared memory.) The Dynamic Server virtual portion of shared memory grows dynamically, but 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 Dynamic Server shared-memory portion so that you can allocate adequate space in the operating system. The amount of space required is the total that all three portions of Dynamic Server shared memory need.

The Resident Portion

The resident portion includes areas of Dynamic Server shared memory that record the state of the database server, including buffers, locks, log files, and the locations of dbspaces, chunks, and tblspaces. The settings that you use for the following Dynamic Server configuration parameters help determine the size of this portion:

- BUFFERS
- LOCKS
- LOGBUFF
- PHYSBUFF

In addition to these parameters, which affect the size of the resident portion, the RESIDENT parameter can affect memory utilization. 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 your version of Dynamic Server indicates whether your operating system supports forced residency.

To estimate the size of the resident portion (in kilobytes) when you allocate operating-system shared memory, take the following steps. The result provides an estimate that slightly exceeds the actual memory used for the resident portion.

To estimate the size of the resident portion

1. To estimate the size of the data buffer, use the following formula:

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

pagesize	is the shared-memory page size, as **oncheck -pr** displays it.

2. Calculate the values in the following formulas:

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

3. To calculate the estimated size of the resident portion in kilobytes, use the following formula:

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

For information about the BUFFERS, LOCKS, LOGBUFF, and PHYSBUFF configuration parameters, see "Configuration Parameters That Affect Memory Utilization" on page 3-31.

Tip: *If you have migrated from Version 4.1 or 5.0 of the database server, you can base your initial estimate of the size of the resident portion on the size of shared memory that output of* ***tbmonitor*** *displays under either of those versions. Because big buffers are not part of the resident portion in Version 6.0 and later, you can deduct 8 pages (8 * pagesize) for each 100 buffers in the BUFFERS parameter listed in the* ***tbconfig*** *file of your earlier version.*

The Virtual Portion

The virtual portion of Dynamic Server shared memory includes the following components:

- Big buffers, which are used for large read and write I/O operations
- Sort-space pools
- Active thread-control blocks, stacks, and heaps
- User session data
- Caches for data-dictionary information and stored procedures
- A global pool for network-interface message buffers and other information

The initial size of the virtual portion is given by the SHMVIRTSIZE configuration parameter in the Dynamic Server configuration file. As the need for additional space in the virtual portion arises, Dynamic Server adds shared memory in increments that the SHMADD configuration parameter specifies, up to a limit on the total shared memory allocated to Dynamic Server that the SHMTOTAL parameter specifies.

The size of the virtual portion is affected primarily by the types of applications and queries that are being run. Depending on your application, 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 4 megabytes if you intend to use data distributions. For guidelines to create data distributions, refer to the discussion of UPDATE STATISTICS in “Creating Data Distributions” on page 10-7.

The basic algorithm to estimate an initial size of the virtual portion of shared memory is as follows:

```
shmvirtsize = fixed overhead + shared structures +
                 (mncs * private structures)
```

To estimate SHMVIRTSIZE with the preceding formula

1. Use the following formula to estimate the fixed overhead:

   ```
   fixed overhead = global pool + thread pool after booting
   ```

   The thread pool after booting is partially dependent on the number of virtual processors.

   Use the **onstat -g mem** command to obtain the pool sizes allocated to sessions.

2. Use the following formula to estimate *shared structures*:

   ```
   shared structures = AIO vectors + sort memory +
                       dbspace backup buffers +
                       dictionary size +
                       size of stored procedure cache +
                       histogram pool +
                       other pools (see onstat display)
   ```

   To see how much memory is allocated to the different pool, use the **onstat -g mem** command.

3. Estimate *mncs* (which is the maximum number of concurrent sessions) with the following formula:

   ```
       mncs = number of poll threads *
                number connections per poll thread
   ```

   The value for *number of poll threads* is the value that you specify in the second field of the NETTYPE configuration parameter.

   The value for *number of connections per poll thread* is the value that you specify in the third field of the NETTYPE configuration parameter.

4. Estimate the private structures with the following formula:

```
private structures = stack + heap +
                        session control-block structures
```

stack	is generally 32 kilobytes but dependent on recursion in stored procedures.
heap	is about 30 kilobytes.
session control-block structures	is the amount of memory that you display can when you use the **onstat -g ses** option.

5. Add the results of steps 1 through 4 to obtain an estimate for SHMVIRTSIZE.

Tip: *When the database server is running with a stable workload, 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.*

The 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 you allow using a given networking 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 using the shared-memory interface, as determined by the NETTYPE parameter for the `ipcshm` protocol.

UNIX

Configuring UNIX Shared Memory

Perform the following steps to configure the shared-memory segments that your Dynamic Server configuration needs. For information on how to set shared-memory-related parameters, refer to the configuration instructions for your operating system.

To configure shared-memory segments for Dynamic Server

1. If your operating system does not have a size limit for shared-memory segments, 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 Dynamic Server configuration requires. This size includes the amount of memory that is required to initialize your Dynamic 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 Dynamic Server.
2. If your operating system has a segment-size limit, 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 Dynamic Server. If there is a remainder, round up to the nearest integer.

```
SHMMNI = total_shmem_size / SHMMAX
```

total_shmem_size	is the total amount of shared memory that you allocate for Dynamic Server use.

3. Set the operating-system configuration parameter for the maximum number of segments, typically SHMMNI, to a value that yields the total amount of shared-memory for Dynamic Server when multiplied by SHMMAX or SHMSIZE. If your computer is dedicated to a single instance of Dynamic Server, that total 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 indicate 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 Dynamic Server.

For additional tips on configuring shared memory in the operating system, refer to the machine-notes file for your version of Dynamic Server.

Using onmode -F to Free Shared Memory

Dynamic Server does not automatically free the shared-memory segments that it adds during its operations. Once memory has been allocated to the Dynamic Server virtual portion, the memory remains unavailable for use by other processes running on the host computer. When Dynamic Server runs a large decision-support query, it might acquire a large amount of shared memory. After the query completes, the database server no longer requires that shared memory. However, the shared memory that the database server allocated to service the query remains assigned to the virtual portion even though it is no longer needed.

The **onmode -F** command locates and returns unused 8-kilobyte blocks of shared memory that Dynamic Server still holds. Although this command runs only briefly (1 or 2 seconds), **onmode -F** dramatically inhibits user activity while it runs. Systems with multiple CPUs and CPU VPs typically experience less degradation while this utility runs.

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

Configuration Parameters That Affect Memory Utilization

The following parameters in the Dynamic Server configuration file have a significant effect on memory utilization:

- SHMVIRTSIZE
- SHMADD
- SHMTOTAL
- BUFFERS
- MAX_RES_BUFFPCT
- RESIDENT
- STACKSIZE
- LOCKS
- LOGBUFF
- PHYSBUFF
- DS_TOTAL_MEMORY
- SHMBASE

In addition, the sizes o f buffers for TCP/IP connections, as specified in the **sqlhosts** file, affect memory and CPU utilization. Sizing these buffers to accommodate a typical request can improve CPU utilization by eliminating the need to break up requests into multiple messages. However, you must use this capability with care; Dynamic Server dynamically allocates buffers of the indicated sizes for active connections. Unless you carefully size buffers, they can consume large amounts of memory.

The SHMBASE parameter indicates the starting address for Dynamic Server shared memory. When set according to the instructions in the machine-notes file for your version of Dynamic Server, this parameter has no appreciable effect on performance.

The following sections describe the performance effects and considerations associated with these parameters. For more information about Dynamic Server configuration parameters, refer to your *Administrator's Guide*.

SHMVIRTSIZE

The SHMVIRTSIZE parameter specifies the size of the virtual portion of shared memory to allocate when you initialize Dynamic Server. The virtual portion of shared memory holds session- and request-specific data as well as other information.

Although Dynamic Server adds increments of shared memory to the virtual portion as needed to process large queries or peak loads, allocation of shared memory increases time for transaction processing. Therefore, Informix recommends that you set SHMVIRTSIZE to provide a virtual portion large enough to cover your normal daily operating requirements.

For an initial setting, Informix suggests that you use the larger of the following values:

- 8,000
- *connections* ∗ 350

The *connections* variable is the number of connections for all network types that are specified in the **sqlhosts** file by one or more NETTYPE parameters. (Dynamic Server uses *connections* ∗ 200 by default.)

Once system utilization reaches a stable workload, you can reconfigure a new value for SHMVIRTSIZE. As noted in "Using onmode -F to Free Shared Memory" on page 3-30, you can instruct Dynamic Server to release shared-memory segments that are no longer in use after a peak workload or large query.

SHMADD

The SHMADD parameter specifies the size of each increment of shared memory that Dynamic Server dynamically adds to the virtual portion. Trade-offs are involved in determining the size of an increment. Adding shared memory consumes CPU cycle. The larger each increment, the fewer increments are required, but less memory is available for other processes. Adding large increments is generally preferred, but when memory is heavily loaded (the scan rate or paging-out rate is high), smaller increments allow better sharing of memory resources among competing programs.

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

Memory Size	SHMADD Value
256 megabytes or less	8,192 kilobytes (the default)
Between 257 and 512 megabytes	16,384 kilobytes
Larger than 512 megabytes	32,768 kilobytes

The size of segments that you add should match those segments allocated in the operating system. For details on configuring shared-memory segments, refer to "Configuring UNIX Shared Memory" on page 3-29. 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. Use the **onstat -g seg** command to display the number of shared-memory segments that Dynamic Server is currently using.

SHMTOTAL

The SHMTOTAL parameter places an absolute upper limit on the amount of shared memory that an instance of Dynamic Server can use. If SHMTOTAL is set to 0 or left unassigned, Dynamic Server continues to attach additional shared memory as needed until no virtual memory is available on the system.

You can usually leave SHMTOTAL set to 0 except in the following cases:

- You must limit the amount of virtual memory that is used by Dynamic Server for other applications or other reasons.
- Your operating system runs out of swap space and performs abnormally.

In the latter case, you can set SHMTOTAL to a value that is a few megabytes less than the total swap space that is available on your computer.

BUFFERS

BUFFERS is the number of data buffers available to Dynamic Server. These buffers reside in the resident portion and are used to cache database data pages in memory.

This parameter has a significant effect on database I/O and transaction throughput. The more buffers that are available, the more likely it is that a needed data page might already reside in memory as the result of a previous request. However, allocating too many buffers can impact the memory-management system and lead to excess paging activity.

Dynamic Server uses the following formula to calculate the amount of memory to allocate for this data buffer pool:

```
bufferpoolsize = BUFFERS * page_size
```

page_size	is the size of a page in memory for your operating system. (Consult the machine-notes file for your version of Dynamic Server for the exact page size.)

Informix suggests that you set BUFFERS between 20 and 25 percent of the number of megabytes in physical memory. For example, if your system has a page size of 2 kilobytes and 100 megabytes of physical memory, you can set BUFFERS to between 10,000 and 12,500, which allocates between 20 megabytes and 25 megabytes of memory.

You can then use **onstat -p** to monitor the read-cache rate. This rate represents the percentage of database pages that are already present in a shared-memory buffer when a query requests a page. (If a page is not already present, the database server must be copy it into memory from disk.) If the database server finds the page in the buffer pool, it spends less time on disk I/O. Therefore, you want a high read-cache rate for good performance.

If the read-cache rate is low, you can repeatedly increase BUFFERS and restart Dynamic 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 the memory-management monitor utility in your operating system, such as **vmstat** or **sar**, to note 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.

RESIDENT

The RESIDENT parameter specifies whether shared-memory residency is enforced for the resident portion of Dynamic Server shared memory; this parameter works only on computers that support forced residency. The resident portion in Dynamic Server contains the least-recently-used (LRU) queues that are used 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, Dynamic 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 temporarily the state of shared-memory residency while Dynamic Server is on-line.
- Change the RESIDENT parameter to turn shared-memory residency on or off the next time that you initialize Dynamic Server shared memory.

STACKSIZE

The STACKSIZE parameter indicates the initial stack size for each thread. Dynamic Server assigns the amount of space that this parameter indicates to each active thread. This space comes from the virtual portion of Dynamic Server shared memory.

To reduce the amount of shared memory that the database server adds dynamically, estimate the amount of the stack space required for the average number of threads that your system runs and include that amount in the value that 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. You can 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 in the NETTYPE parameters in your ONCONFIG file), depending on your workload.

LOCKS

The LOCKS parameter sets the maximum number of locks that you can use at any one time. The absolute maximum number of locks in Dynamic Server is 8 million. Each lock requires 44 bytes in the resident segment. You must provide for this amount of memory when you configure shared memory.

Set LOCKS to the maximum number of locks that a query needs, 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 Statement	Isolation Level	Table	Row	Key	BYTE or TEXT Data
SELECT	Dirty Read	0	0	0	0
	Committed Read	1	0	0	0
	Cursor Stability	1	1	0	0
	Indexed Repeatable Read	1	Number of rows that satisfy conditions	Number of rows that satisfy conditions	0
	Sequential Repeatable Read	1	0	0	0
INSERT		1	1	Number of indexes	Number of pages in BYTE or TEXT data
DELETE		1	1	Number of indexes	Number of pages in BYTE or TEXT data
UPDATE		1	1	2 per changed key value	Number of pages in old plus new BYTE or TEXT data

Important: *During the execution of the SQL statement DROP DATABASE, Dynamic Server acquires and holds a lock on each table in the database until the entire DROP operation completes. Make sure that your value for LOCKS is large enough to accommodate the largest number of tables in a database.*

LOGBUFF

The LOGBUFF parameter determines 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.

PHYSBUFF

The PHYSBUFF parameter determines 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. Choose a value for PHYSBUFF that is an even increment of the system page size.

DS_TOTAL_MEMORY

The DS_TOTAL_MEMORY parameter places a ceiling on the amount of shared memory that a query can obtain. You can use this parameter to limit the performance impact of large, memory-intensive queries. The higher you set this parameter, the more memory a large query can use, and the less memory is available for processing other queries and transactions.

For OLTP applications, set DS_TOTAL_MEMORY to between 20 and 50 percent of the value of SHM_TOTAL, in kilobytes. For applications that involve large decision-support (DSS) queries, increase the value of DS_TOTAL_MEMORY to between 50 and 80 percent of SHM_TOTAL. If you use your Dynamic Server instance exclusively for DSS queries, set this parameter to 90 percent of SHM_TOTAL.

A *quantum unit* is the minimum increment of memory allocated to a query. The Memory Grant Manager (MGM) allocates memory to queries in quantum units. Dynamic Server uses the value of DS_MAX_QUERIES with the value of DS_TOTAL_MEMORY to calculate a quantum of memory, according to the following formula:

```
quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

To allow for more simultaneous queries with smaller quanta each, Informix suggests that you increase DS_MAX_QUERIES. For more information on DS_MAX_QUERIES, refer to "DS_MAX_QUERIES" on page 3-16. For more information on MGM, refer to "The Memory Grant Manager" on page 9-5.

Algorithm for Determining DS_TOTAL_MEMORY

Dynamic Server derives a value for DS_TOTAL_MEMORY if you do not set DS_TOTAL_MEMORY or if you set it to an inappropriate value. Whenever Dynamic Server changes the value that you assigned to DS_TOTAL_MEMORY, it sends the following message to your console:

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

The metavariable *old_value* represents the value that you assigned to DS_TOTAL_MEMORY in your configuration file. The metavariable *new_value* represents the value that Dynamic Server derived.

When you receive the preceding message, you can use the algorithm to investigate what values Dynamic Server considers inappropriate. You can then take corrective action based on your investigation.

The following sections documents the algorithm that Dynamic Server uses to derive the new value for DS_TOTAL_MEMORY.

Derive a Minimum for Decision-Support Memory

In the first part of the algorithm, Dynamic Server establishes a minimum for decision- support memory. When you assign a value to the configuration parameter DS_MAX_QUERIES, Dynamic Server sets the minimum amount of decision-support memory according to the following formula:

```
min_ds_total_memory = DS_MAX_QUERIES * 128Kb
```

When you do not assign a value to DS_MAX_QUERIES, Dynamic Server instead uses the following formula, which is based on the value of NUMCPU-VPS:

```
min_ds_total_memory = NUMCPUVPS * 2 * 128Kb
```

Derive a Working Value for Decision-Support Memory

In the second part of the algorithm, Dynamic Server establishes a working value for the amount of decision-support memory. Dynamic Server verifies this amount in the third and final part of the algorithm.

When DS_TOTAL_MEMORY Is Set

Dynamic Server first checks if SHMTOTAL is set. When SHMTOTAL is set, Dynamic 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 effectively prevents you from setting DS_TOTAL_MEMORY to values that Dynamic Server cannot possibly allocate to decision-support memory.

When SHMTOTAL is not set, Dynamic Server sets decision-support memory equal to the value that you specified in DS_TOTAL_MEMORY.

When DS_TOTAL_MEMORY Is Not Set

When you do not set DS_TOTAL_MEMORY, Dynamic Server proceeds as follows. First, Dynamic Server checks if you have set SHMTOTAL. When SHMTOTAL is set, Dynamic Server uses the following formula to calculate the amount of decision-support memory:

```
decision_support_memory = SHMTOTAL -
                          nondecision_support_memory
```

When Dynamic Server finds that you did not set SHMTOTAL, it sets decision-support memory as the following example shows:

```
decision_support_memory = min_ds_total_memory
```

For a description of the variable **min_ds_total_memory**, refer to "Derive a Minimum for Decision-Support Memory" on page 3-40.

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 memory space for your computer. When Dynamic 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 Dynamic Server finds that the derived value for decision-support memory is greater than the maximum possible memory space for your computer, it sets decision-support memory equal to the maximum possible memory space.

Data-Replication Buffers and Memory Utilization

Data replication requires two instances of Dynamic Server, a primary one and a secondary one, running on two computers. If you implement data replication for your Dynamic Server database server, Dynamic Server holds logical-log records in the data-replication buffer before it sends them to the secondary Dynamic Server. The data-replication buffer is always the same size as the logical-log buffer.

Memory-Resident Tables and the Buffer Pool

If your database contains smaller tables that are used by applications for which performance is important, you can instruct the database server to leave data and index pages for these tables in the buffer cache. A table whose pages should remain in memory is known as a *memory-resident* table.

Although the least recently used mechanism tends to keep pages that the database server reads often in memory, it does not improve performance in cases where smaller tables are not accessed often enough to be kept in memory. For more information on the least recently used mechanism, refer to your *Administrator's Guide*.

You specify which tables or fragments should be memory-resident with the SET TABLE statement in SQL. You must execute this statement every time that you bring up the database server. For more information about the SET TABLE statement, refer to the *Informix Guide to SQL: Syntax*.

You can also specify that indexes be memory resident with the SET INDEX statement in SQL.

Use caution when you determine which tables, fragments, or indexes should be memory resident. If there are too many memory-resident tables, or if the memory-resident tables are too large, the database server might swap out pages that do not belong to memory-resident tables too often. An improper use of memory-resident tables can decrease performance for applications that use tables that are not set to memory resident.

You can monitor the effect of memory-resident tables with the following **onstat** options:

- **onstat -P**
- **onstat -b**
- **onstat -B**
- **onstat -p**

You can list the number of buffer pages that have data or index pages from memory-resident tables with **onstat -P**. The following excerpt from an **onstat -P** output shows three tables, fragments, or indexes that have a total of four memory-resident pages:

```
partnum  total    btree    data     other    resident
0        99       17       21       61       0
1048578  3        1        1        1        0
1048579  6        4        2        0        0
...
1048828  2        1        0        1        2
1048829  1        0        0        1        1
1048830  1        0        0        1        1
```

Also, the **onstat -b** (or **onstat -B**) command shows the total number of memory resident pages at any one time. The following **onstat -b** output shows that there are four memory-resident pages out of a total of 200 pages in the buffer pool:

```
Buffers
address  userthread flgs pagenum  memaddr  nslots pgflgs xflgs
owner    waitlist
 0 modified, 4 resident, 200 total, 256 hash buckets, 2048
buffer size
```

As usual, you should monitor the buffer cache rate with **onstat -p**, as the cache rate can decrease significantly when memory-resident tables are used improperly.

How Configuration Affects I/O Activity

Your Dynamic Server configuration affects I/O activity in several ways. Your assignment of chunks and dbspaces can create I/O *hot spots*, or disk partitions with a disproportionate amount of I/O activity. Your allocation of critical data, sort areas, and areas for temporary files and index builds can place intermittent loads on various disks. How you configure read-ahead can increase the effectiveness of individual I/O operations. How you configure the background I/O tasks, such as logging and page cleaning, can affect I/O throughput. The following sections discuss each of these topics.

Chunk and Dbspace Configuration

All the data that resides in a Dynamic Server database is stored on disk. Optical Subsystem also uses a magnetic disk to access BYTE or TEXT data that is retrieved from optical media. The speed at which Dynamic Server can copy the appropriate data pages to and from disk determines how well your application performs.

Disks are typically 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 typically outside the control of the Dynamic Server administrator.

Disks can become overused or saturated when users request pages too often. Saturation can occur in the following situations:

- You use a disk for multiple purposes, such as for both logging and active database tables.
- Disparate data resides on the same disk.
- Table extents become interleaved.

The various functions that your application requires, as well as the consistency-control functions that Dynamic Server performs, determine the optimal disk, chunk, and dbspace layout for your application. The more disks that you make available to Dynamic Server, the easier it is to balance I/O across them. For more information on these factors, refer to Chapter 4.

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

- Placement and mirroring of critical data
- Load balancing
- Reduction of contention
- Ease of backup and restore

Associate Disk Partitions with Chunks

Informix recommends that you assign chunks to entire disk partitions. When a chunk coincides with a disk partition (or device), it is easy to track disk-space use, and you avoid errors caused by miscalculated offsets.

Associate Dbspaces with Chunks

In Version 5.0 and earlier of the database server, you can improve performance when you combine several chunks within a dbspace. In versions later than 5.0, this is no longer the case. Informix recommends that you associate a single chunk with a dbspace, especially when that dbspace is to be used for a table fragment. For more information on table placement and layout, refer to Chapter 4.

Place Database System Catalog Tables with Database Tables

When a disk that contains the system catalog for a particular database fails, the entire database remains inaccessible until the catalog is restored. Because of this potential inaccessibility, 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.

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 statements DATABASE or CONNECT to make that database the current database.
3. Enter the CREATE TABLE statement to create the table.

Placement 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 Dynamic Server. Dynamic Server cannot operate if any of these elements becomes unavailable. By default, Dynamic Server places all three critical elements in the root dbspace.

To arrive at an appropriate placement strategy for critical data, you must make a trade-off between protecting the availability of data and allowing maximum logging performance.

Dynamic Server also places temporary table and sort files in the root dbspace by default. Informix recommends that you use the DBSPACETEMP configuration parameter and the **DBSPACETEMP** environment variable to assign these tables and files to other dbspaces. For details, see "Parameters and Variables That Affect Temporary Tables and Sorting" on page 3-51.

Consider Separate Disks for Critical Data Components

If you place the root dbspace, logical log, and physical log in separate dbspaces on separate disks, you can obtain some distinct performance advantages. The disks that you use for each critical data component should be on separate controllers. This approach has the following advantages:

- Isolates logging activity from database I/O and allows physical-log I/O requests to be serviced in parallel with logical-log I/O requests
- Reduces the time that you need to recover from a crash

  However, unless the disks are mirrored, there is an increased risk that a disk that contains critical data might be affected in the event of a crash, which will bring Dynamic Server to a halt and require the complete restoration of all data from a level-0 backup.
- Allows for 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.

Dynamic Server uses different methods to configure various portions of critical data. To assign an appropriate dbspace for the root dbspace and physical log, set the appropriate Dynamic Server configuration parameters. To assign the logical-log files to an appropriate dbspace, use the **onparams** utility.

For more information on the configuration parameters that affect each portion of critical data, refer to "Configuration Parameters That Affect Critical Data" on page 3-49.

Consider Mirroring for Critical Data Components

Consider mirroring for the dbspaces that contain critical data. Mirroring these dbspaces ensures that Dynamic Server can continue to operate even when a single disk fails. However, depending on the mix of I/O requests for a given dbspace, a trade-off exists between the fault tolerance of mirroring and I/O performance. You obtain a marked performance advantage when you mirror dbspaces that have a read-intensive usage pattern and a slight performance disadvantage when you mirror write-intensive dbspaces.

When mirroring is in effect, two disks are available to handle read requests, and Dynamic Server can process a higher volume of those requests. However, each write request requires two physical write operations and does not complete until both physical operations are performed. The write operations are performed in parallel, but the request does 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 more update-intensive in other, nonmirrored dbspaces, you can use the Dynamic Server archive-and-backup facilities to perform warm restores of those tables in the event of a disk failure. When the root dbspace is mirrored, Dynamic 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 MIRRORPATH configuration parameter should have a different value than 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 doubled-write performance penalty noted in "Consider Mirroring for Critical Data Components" on page 3-47. However, you can adjust the rate at which logging generates I/O requests to a certain extent by choosing an appropriate log buffer size and logging mode.

With unbuffered and ANSI-compliant logging, Dynamic Server requests a flush of the log buffer to disk for every committed transaction (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, buffered logging leaves transactions in any partially filled buffers vulnerable to loss in the event of a system failure.

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 that you might need to reenter when service is restored after a fault.

Unlike the physical log, you cannot specify an alternative dbspace for logical-log files in your initial Dynamic Server configuration. Instead, use the **onparams** utility first to add logical-log files to an alternative dbspace and then drop logical-log files from the root dbspace. For more information about **onparams**, refer to your *Administrator's Guide*.

Mirroring the Physical Log

The physical log is write intensive, with activity occurring at checkpoints and when buffered data pages are flushed. 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 doubled-write performance penalty noted under "Consider Mirroring for Critical Data Components" on page 3-47. You can adjust your checkpoint interval (see "CKPINTVL" on page 3-65) and your LRU minimum and maximum thresholds (see "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 3-68) to keep I/O to the physical log at a minimum.

Configuration Parameters That Affect Critical Data

You can use the following parameters to configure the root dbspace:

- ROOTNAME
- ROOTOFFSET
- ROOTPATH
- ROOTSIZE

- MIRROR
- MIRRORPATH
- MIRROROFFSET

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). Otherwise, these parameters have no major impact on performance.

The following Dynamic Server configuration parameters affect the logical logs:

- LOGSIZE
- LOGSMAX
- LOGBUFF

LOGSIZE and LOGSMAX determine the size and number of logical-log files. 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 3-38.

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

- PHYSDBS
- PHYSFILE

Dbspaces for Temporary Tables and Sort Files

If your applications use temporary tables or large sort operations, you can improve performance by using the DBSPACETEMP configuration parameter to designate one or more dbspaces for temporary tables and sort files. When you specify more than one dbspace for temporary tables, Dynamic Server automatically applies its parallel insert capability to fragment the temporary table across those dbspaces, using a round-robin distribution scheme. When you assign two or more dbspaces on separate disks for temporary tables, you can dramatically improve the speed with which Dynamic Server creates those tables.

Applications can designate dbspaces for temporary tables and sort files with the **DBSPACETEMP** environment variable, as described in "Parameters and Variables That Affect Temporary Tables and Sorting" on page 3-51. You can also create special dbspaces, called *temporary dbspaces*, to be used exclusively to store temporary tables and sort files.

To create a dbspace for the exclusive use of temporary tables and sort files, use **onspaces -t**. If you create more than one temporary dbspace, each dbspace should reside on a separate disk to balance the I/O impact. For best performance, place no more than one temporary dbspace on a single disk.

Both logical and physical logging are suppressed for temporary dbspaces, and these dbspaces are never backed up as part of a full-system backup. You cannot mirror a temporary dbspace that you create with **onspaces -t**.

Parameters and Variables That Affect Temporary Tables and Sorting

The DBSPACETEMP configuration parameter affects the placement of temporary tables and sort files, as does the **DBSPACETEMP** environment variable.

The DBSPACETEMP Configuration Parameter

The DBSPACETEMP configuration parameter specifies a list of dbspaces in which Dynamic Server places temporary tables and sort files by default. If you specify more than one dbspace in this list, Dynamic Server uses its parallel insert capability to fragment temporary tables across all the listed dbspaces, using a round-robin distribution scheme. For more information, refer to "Designing a Distribution Scheme" on page 6-12.

Important: *For best performance, Informix recommends that you use DBSPACETEMP to specify two or more dbspaces on separate disks for temporary tables and sort files.*

Some or all of the dbspaces that you list in this parameter can be temporary dbspaces, which are reserved exclusively to store temporary tables and sort files. For details, see "Dbspaces for Temporary Tables and Sort Files" on page 3-50.

To override the DBSPACETEMP parameter, you can use the **DBSPACETEMP** environment variable for both temporary tables and sort files. Although you can use the **PSORT_DBTEMP** environment variable to specify one or more operating-system directories in which to place sort files, you can obtain better performance when they specify dbspaces in **DBSPACETEMP**. When dbspaces reside on unbuffered devices, I/O is faster to those devices than to files buffered through the operating system.

Important: *Informix recommends that you use the DBSPACETEMP parameter or the* ***DBSPACETEMP*** *environment variable for better performance of sort operations and to prevent Dynamic Server from unexpectedly filling file systems.*

When you do not specify dbspaces in the DBSPACETEMP configuration parameter or the client **DBSPACETEMP** environment variable, Dynamic Server places all temporary tables in the root dbspace. This action can severely affect I/O to the root dbspace. If the root dbspace is mirrored, you encounter a slight doubled-write performance penalty for I/O to the temporary tables and sort files. In addition, Dynamic Server places explicit temporary tables created with the TEMP TABLE clause of the CREATE TABLE statement in the dbspace of the current database. This action can have severe impacts on I/O to that dbspace.

The DBSPACETEMP and PSORT_NPROCS Environment Variables

The **DBSPACETEMP** environment variable is similar to the DBSPACETEMP configuration parameter that the preceding section describes. When set in the environment for a client application, this variable overrides the DBSPACETEMP configuration parameter.

This environment variable specifies a list of one or more dbspaces in which to place temporary tables for a particular session. To balance the load of temporary tables, Dynamic Server cycles through this list to distribute temporary tables among the specified dbspaces. For best performance, do not list more than one dbspace on any given disk in **DBSPACETEMP**.

For the following reasons, Informix recommends that you use **DBSPACETEMP** rather than the **PSORT_DBTEMP** environment variable to specify sort files:

- **DBSPACETEMP** typically yields better performance. When dbspaces reside on unbuffered devices, I/O is faster to those devices than to buffered files.
- **PSORT_DBTEMP** specifies one or more operating-system directories in which to place sort files. Therefore, Dynamic Server can unexpectedly fill file systems on your computer.

The database server writes sort files to the **/tmp** directory of the operating system when you do not specify a location with the DBSPACETEMP configuration parameter, the client **DBSPACETEMP** environment variable, or the client **PSORT_DBTEMP** environment variable. If the file system that contains that directory has insufficient space to hold a sort file, the query performing the sort returns an error. Meanwhile, the operating system might be severely impacted until you remove the sort file.

The **PSORT_NPROCS** environment variable specifies the maximum number of threads that the database server can use to sort a query. When the value of PDQ priority is 0 and **PSORT_NPROCS** is greater than 1, Dynamic Server uses parallel sorts. These sorts are not limited by the management of PDQ. In other words, although the sort is executed in parallel, Dynamic Server *does not* regard sorting as a PDQ activity. Dynamic Server does not control sorting by any of the PDQ configuration parameters when PDQ priority is 0.

When PDQ priority is greater than 0 and **PSORT_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable to request a specific proportion of PDQ resources for a query. You can use the MAX_PDQPRIORITY parameter to limit the number of such user requests. For more information on MAX_PDQPRIORITY, refer to "MAX_PDQPRIORITY" on page 3-15.

Dynamic Server allocates a relatively small amount of memory for sorting, and that memory is divided among the **PSORT_NPROCS** sort threads. For more information on memory allocated for sorting, refer to "Estimating Sort Memory" on page 4-28.

Important: *For better performance for a sort operation, Informix recommends that you set **PSORT_NPROCS** initially to 2 when your computer has multiple CPUs. If the subsequent CPU activity is lower than I/O activity, you can increase the value of **PSORT_NPROCS**.*

For more information about environment variables that users can set, refer to the *Informix Guide to SQL: Reference*.

How Blobspace Configuration Affects Performance

A blobspace is a logical storage unit composed of one or more chunks that store only BYTE and TEXT data. If you use a blobspace, you can store BYTE or TEXT data on a separate disk from the table with which the data is associated. You can store BYTE or TEXT data associated with different tables in the same blobspace.

To create a blobspace, use ON-Monitor or the **onspaces** utility, as your *Administrator's Guide* describes. You assign BYTE or TEXT data to a blobspace when you create the tables with which the BYTE or TEXT data is associated. For more information on the SQL statement CREATE TABLE, refer to the *Informix Guide to SQL: Syntax*.

When you store BYTE or TEXT data in a blobspace on a separate disk from the table with which it is associated, Dynamic Server provides the following performance advantages:

- Parallel access to table and BYTE or TEXT data
- Unlike BYTE or TEXT data stored in a tblspace, blobspace data is written directly to disk. This data is not logged, which reduces logging I/O activity for logged databases. The BYTE or TEXT data does not pass through resident shared memory, which leaves memory pages free for other uses.

Blobspaces are divided into units called *blobpages*. Dynamic Server retrieves BYTE or TEXT data from a blobspace in blobpage-sized units.You specify the size of a blobpage in multiples of a disk page when you create the blobspace. The optimal blobpage size for your configuration depends on the following factors:

- The size distribution among the BYTE or TEXT data
- The trade-off between retrieval speed for your largest data and the amount of disk space that is wasted by storing small amounts of BYTE or TEXT data in large blobpages

To retrieve BYTE or TEXT data as quickly as possible, use the size of your largest BYTE or TEXT data rounded up to the nearest disk-page-sized increment. This scheme guarantees that Dynamic Server can retrieve even the largest BYTE or TEXT data in a single I/O request. Although this scheme guarantees the fastest retrieval, it has the potential to waste disk space. Because BYTE or TEXT data is stored in its own blobpage (or set of blobpages), Dynamic Server reserves the same amount of disk space for every blobpage even if the BYTE or TEXT data takes up a fraction of that page. Using a smaller blobpage allows you to make better use of your disk, especially when you have large differences in the sizes of your BYTE or TEXT data.

To achieve the greatest theoretical utilization of space on your disk, you could make your blobpage the same size as a standard disk page. Then, much , if not most, BYTE or TEXT data would require several blobpages. Because Dynamic Server acquires a lock and issues a separate I/O request for each blobpage, this scheme performs poorly.

In practice, a balanced scheme for sizing uses the most frequently occurring BYTE or TEXT data size as the size of a blobpage. For example, you have 160 TEXT or BYTE data values in a table, 120 values are 12 kilobytes each and 40 columns are 16 kilobytes each. Then a 12-kilobyte blobpage size provides greater storage efficiency than a 16-kilobyte blobpage size. This configuration allows the majority of TEXT or BYTE columns to require a single blobpage and the other 40 columns to require two blobpages. In this configuration, 8 kilobytes are wasted in the second blobpage for each of the larger values.

Tip: *If a table has more than one BYTE or TEXT data columns and the data values are not close in size, store the data in different blobspaces, each with an appropriately sized blobpage.*

How Optical Subsystem Affects Performance

Optical Subsystem extends the storage capabilities of Dynamic Server for BYTE or TEXT data to write-once-read-many (WORM) optical subsystems. Dynamic Server uses a cache in memory to buffer initial BYTE or TEXT data pages requested from the optical subsystem. The memory cache is a common storage area. Dynamic Server adds BYTE or TEXT data requested by any application to the memory cache as long as the cache has space. To free space in the memory cache, the application must release the BYTE or TEXT data that it is using.

A significant performance advantage occurs when you retrieve BYTE or TEXT data directly into memory instead of buffering that data on disk. Therefore, proper cache sizing is important when you use Optical Subsystem. You specify the total amount of space available in the memory cache with the OPCACHEMAX configuration parameter. Applications indicate that they require access to a portion of the memory cache when they set the **INFORMIXOPCACHE** environment variable. For details, refer to "INFORMIXOPCACHE" on page 3-58.

BYTE or TEXT data that does not fit entirely into the space that remains in the cache are stored in the blobspace that the STAGEBLOB configuration parameter names. This staging area acts as a secondary cache on disk for blobpages that are retrieved from the optical subsystem. BYTE or TEXT data that is retrieved from the optical subsystem is held in the staging area until the transactions that requested it are complete.

The Dynamic Server administrator creates the staging-area blobspace with ON-Monitor or with the **onspaces** utility.

You can use **onstat -O** to monitor utilization of the memory cache and STAGEBLOB blobspace. If contention develops for the memory cache, increase the value listed in the configuration file for OPCACHEMAX. (The new value takes effect the next time that Dynamic Server initializes shared memory.) For a complete description of Optical Subsystem, refer to the *Guide to the Optical Subsystem*.

Environment Variables and Configuration Parameters Related to Optical Subsystem

The following configuration parameters affect the performance of Optical Subsystem:

- STAGEBLOB
- OPCACHEMAX

The following sections describe these parameters, along with the **INFORMIXOPCACHE** environment variable.

STAGEBLOB

The STAGEBLOB configuration parameter identifies the blobspace that is to be used as a staging area for BYTE or TEXT data that is retrieved from the optical subsystem, and it activates Optical Subsystem. If the STAGEBLOB parameter is not listed in the configuration file, Optical Subsystem does not recognize the optical-storage subsystem.

The structure of the staging-area blobspace is the same as all other Dynamic Server blobspaces. When the Dynamic Server administrator creates the staging area, it consists of only one chunk, but you can add more chunks as desired. You cannot mirror the staging-area blobspace. The optimal size for the staging-area blobspace depends on the following factors:

- The frequency of BYTE or TEXT data storage
- The frequency of BYTE or TEXT data retrieval
- The average size of the BYTE or TEXT data to be stored

To calculate the size of the staging-area blobspace, you must estimate the number of blobs that you expect to reside there simultaneously and multiply that number by the average BYTE or TEXT data size.

OPCACHEMAX

The OPCACHEMAX configuration parameter specifies the total amount of space that is available for BYTE or TEXT data retrieval in the memory cache that Optical Subsystem uses. Until the memory cache fills, it stores BYTE or TEXT data that is requested by any application. BYTE or TEXT data that cannot fit in the cache is stored on disk in the blobspace that the STAGEBLOB configuration parameter indicates. You can increase the size of the cache to reduce contention among BYTE or TEXT data requests and to improve performance for requests that involve Optical Subsystem.

INFORMIXOPCACHE

The **INFORMIXOPCACHE** environment variable sets the size of the memory cache that a given application uses for BYTE or TEXT data retrieval. If the value of this variable exceeds the maximum that the OPCACHEMAX configuration parameter specifies, OPCACHEMAX is used instead. If **INFORMIXOPCACHE** is not set in the environment, the cache size is set to OPCACHEMAX by default.

I/O for Tables

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

Sequential Scans

When Dynamic Server performs a sequential scan of data or index pages, most of the I/O wait time is caused by seeking the appropriate starting page. You can dramatically improve performance for sequential scans by bringing in a number of contiguous pages with each I/O operation. The action of bringing additional pages along with the first page in a sequential scan is called *read-ahead*.

The timing of I/O operations that are needed for a sequential scan is also important. If the scan thread must wait for the next set of pages to be brought in after working 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, depends 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 and RA_THRESHOLD" on page 3-60.

Light Scans

Under the appropriate circumstances, Dynamic Server can bypass the LRU queues when it performs a sequential scan. A sequential scan that avoids the LRU queues is termed a *light scan*. Light scans can be used only for sequential scans of large data tables and are the fastest means for performing these scans. System catalog tables and tables smaller than the size of the buffer pool do not use light scans. Light scans are allowed under Dirty Read (including nonlogging databases) and Repeatable Read isolation levels. Repeatable Read full-table scans obtain a shared lock on the table. A light scan is used only in Committed Read isolation if the table has a shared lock. Light scans are never allowed under Cursor Stability isolation.

Unavailable Data

Another aspect of I/O for tables has to do with situations in which a query requests access to a table or fragment in a dbspace that is temporarily unavailable. When Dynamic Server determines that a dbspace is unavailable as the result of a disk failure, queries directed to that dbspace fail by default. Dynamic Server allows you to specify dbspaces that, when unavailable, can be skipped by queries, as described in "DATASKIP" on page 3-61.

Warning: *If a dbspace that contains data that a query requests is listed in DATASKIP and is currently unavailable because of a disk failure, the data that Dynamic Server returns to the query can be inconsistent with the actual contents of the database.*

Configuration Parameters That Affect I/O for Tables

The following configuration parameters affect read-ahead:

- RA_PAGES
- RA_THRESHOLD

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

The following sections describes the performance effects and considerations that are associated with these parameters. For more information about Dynamic Server configuration parameters, refer to your *Administrator's Guide*.

RA_PAGES and RA_THRESHOLD

The RA_PAGES parameter indicates the number of pages that Dynamic Server brings into memory in a single I/O operation during sequential scans of data or index pages. The RA_THRESHOLD parameter indicates the point at which Dynamic Server issues an I/O request to bring in the next set of pages from disk. Because the greater portion of I/O wait time is involved in seeking the correct starting point on disk, you can increase efficiency of sequential scans by increasing the number of contiguous pages brought in with each transfer. However, setting RA_PAGES too large or RA_THRESHOLD too high with respect to BUFFERS can trigger unnecessary page cleaning to make room for pages that are not needed immediately.

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. If you want to allow large scans to take up to 75 percent of buffers, *bp_fract* would be 0.75.
large_queries	is the number of concurrent queries that require read-ahead that you intend to support.

DATASKIP

The DATASKIP parameter allows you to specify which dbspaces, if any, queries can skip when those dbspaces are unavailable as the result of a disk failure. You can list specific dbspaces and turn data skipping on or off for all dbspaces. For details, refer to your *Administrator's Guide*.

Dynamic 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: *Dynamic Server cannot determine whether the results of a query are consistent when a dbspace is skipped. If the dbspace contains a table fragment, the user who executes the query must ensure that the rows within that fragment are not needed for an accurate query result. Turning DATASKIP on allows queries with incomplete data to return results that can be inconsistent with the actual state of the database. Without proper care, that data can yield incorrect 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 Dynamic Server operation. However, they create overhead in the CPU and take up I/O bandwidth. These overhead activities take 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 your application.

The following list shows some background I/O activities:

- Checkpoints
- Logging
- Page cleaning
- Backup and restore
- Rollback and recovery
- Data replication
- Auditing

Checkpoints occur regardless of whether there is much database activity, although they can occur with greater frequency as activity increases. Other background activities, such as logging and page cleaning, occur more frequently 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. A direct trade-off exists between the frequency of checkpoints or the size of the logical logs and the time that it takes to recover the database in the event of a failure. So a major consideration when you attempt 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 be unable to find the available pages that it needs. It must then initiate a *foreground write* and wait for pages to be freed. Foreground writes impair performance, so you should avoid them. 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 3-68.) Use **onstat -F** to monitor the frequency of foreground writes.

For the most part, tuning your 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 shift the performance bottleneck to another.

Data replication typically adds overhead to logical-log I/O that is proportional to your logging activity. A configuration that optimizes logging activity without data replication is likely to optimize logging with data replication.

The effect of auditing on performance is largely determined by the auditing events that you choose to record. Depending on which users and events are audited, the impact of this feature can vary widely. Infrequent events, such as requests to connect to a database, have low performance impact. Frequent events, such as requests to read any row, can generate a large amount of auditing activity. The more users for whom such frequent events are audited, the greater the impact on performance. For information about auditing, refer to your *Trusted Facility Manual*.

Data replication and auditing are optional. You can obtain immediate performance improvements by disabling these features, provided that the operating requirements for your system allow you to do so.

Configuration Parameters That Affect Background I/O

The following configuration parameters affect checkpoints:

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

The following configuration parameters affect logging:

- LOGBUFF
- PHYSBUFF
- LTXHWM
- LTXEHWM
- LBU_PRESERVE

The following configuration parameters affect page cleaning:

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

"RA_PAGES and RA_THRESHOLD" on page 3-60 describes the RA_PAGES and RA_THRESHOLD parameters.

The following configuration parameters affect backup and restore:

- TAPEBLK
- LTAPEBLK
- TAPEDEV
- LTAPEDEV
- TAPESIZE
- LTAPESIZE

The following configuration parameters affect fast recovery:

- OFF_RECVRY_THREADS
- ON_RECVRY_THREADS

The following configuration parameters affect data-replication performance:

- DRINTERVAL
- DRTIMEOUT

The following configuration parameters affect auditing performance:

- ADTERR
- ADTMODE

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

CKPINTVL

The CKPTINTVL parameter specifies the maximum interval between checkpoints. Dynamic Server can skip a checkpoint if all data is physically consistent when the checkpoint interval expires. Dynamic Server writes a message to the message log that notes 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. If you set CKPTINTVL to a long interval, you can use physical-log capacity to trigger checkpoints based on actual database activity instead of an arbitrary time unit. However, a long checkpoint interval can increase the time needed for recovery in the event of a failure. 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, depending on physical-logging activity.

LOGSIZE, LOGFILES, LOGSMAX, and PHYSFILE

The LOGSIZE parameter indicates the size of the logical log. You can 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 by one or more NETTYPE parameters.
maxrows	is the largest number of rows to be updated in a single transaction.

To obtain better overall performance for applications that perform frequent updates of BYTE or TEXT data in blobspaces, reduce the size of the logical log. Blobpages cannot be reused until the logical log to which they are allocated is backed up. When BYTE or TEXT data activity is high, the performance impact of more-frequent checkpoints is balanced by the higher availability of free blobpages.

LOGSIZE, LOGFILES, and LOGSMAX indirectly affect checkpoints because they specify the size and number of logical-log files. A checkpoint can occur when Dynamic 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. (See "LTXHWM and LTXEHWM" on page 3-67.) The log should be large enough to accommodate the longest transaction you are likely to encounter that is not the result of an error.

When you use volatile blobpages in blobspaces, smaller logs can improve access to BYTE or TEXT data that must be reused. BYTE or TEXT data cannot be reused until the log in which it is allocated is flushed to disk. In this case, you can justify the cost in performance because those smaller log files are backed up more frequently.

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 is update intensive and updates tend not to occur on the same pages, you can use the following formula to calculate the size of the physical log:

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

This value represents a maximum. You can reduce the size of the physical log when applications are less update intensive or when updates tend to cluster within the same pages. If you increase the checkpoint interval (see "CKPINTVL" on page 3-65) or anticipate increased activity, consider increasing the size of the physical log. You can decrease the size of the physical log if you intend to use physical-log fullness to trigger checkpoints.

ONDBSPDOWN

The ONDBSPDOWN parameter specifies the response that Dynamic Server takes when an I/O error indicates that a dbspace is down. By default, Dynamic Server marks any dbspace that contains no critical data as `down` and continues processing. Critical data include the root dbspace, the logical log, or the physical log. To restore access to that database, you must back up all logical logs and then perform a warm restore on the down dbspace.

Dynamic 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, Dynamic Server continues processing to the next checkpoint and then suspends processing of all update requests. Dynamic Server repeatedly retries the I/O request that produced the error until the dbspace has been repaired and the request completes or the Dynamic 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: *This 2 setting for ONDBSPDOWN can impact the performance for update requests severely because they are suspended due to a down dbspace. When you use this setting for ONDBSPDOWN, be sure to monitor the status of your dbspaces.*

When ONDBSPDOWN is set to 1, Dynamic 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 and performing a cold restore when any dbspace goes down can be severe.

Important: *Consider mirroring all your dbspaces if you decide to set ONDBSPDOWN to 1.*

LOGBUFF and PHYSBUFF

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

LTXHWM and LTXEHWM

The LTXHWM and LTXEHWM parameters specify the maximum limits for long transactions and long-transaction exclusive rollbacks. The LTXHWM parameter indicates how full a logical log is when Dynamic Server starts to check for a possible long transaction. The LTXEHWM parameter indicates the point at which Dynamic Server suspends new transaction activity to locate and roll back a long transaction. These events should be rare, but if they occur, they can 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. (See “LOGSIZE, LOGFILES, LOGSMAX, and PHYSFILE” on page 3-65.)

LBU_PRESERVE

The LBU_PRESERVE parameter, when set to a value of 1, suspends ordinary transaction processing whenever the second-to-last logical-log file reaches its maximum limit. This parameter is useful in preventing long transaction errors when continuous backup is in effect. A long transaction can occur when ON-Archive attempts to add log records while it backs up the logical log. When LBU_PRESERVE is not in effect, and the transaction rate is high, the last log can fill before the backup operation completes. Informix recommends that you use LBU_PRESERVE and add an extra logical-log file whenever you activate continuous backup with ON-Archive. For details, refer to your *Archive and Backup Guide*.

LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY

The LRUS parameter indicates 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.

You use LRUS 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 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 **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.

CLEANERS

The CLEANERS parameter indicates 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 Dynamic 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 previous section describes, increase the number of page-cleaner threads proportionally.

TAPEBLK, TAPEDEV, TAPESIZE, LTAPEBLK, LTAPEDEV, and LTAPESIZE

The TAPEBLK parameter specifies the block size for database backups made with **ontape**, **onload**, and **onunload**. TAPEDEV specifies the tape device. TAPESIZE specifies the tape size for these backups. The LTAPEBLK, LTAPEDEV, and LTAPESIZE parameters specify the block size, device, and tape size for logical-log backups made with **ontape**. For information about these utilities, the configuration procedures for ON-Archive, and specific recommendations for backup-and-restore operations, refer to your *Archive and Backup Guide*.

OFF_RECVRY_THREADS and ON_RECVRY_THREADS

The OFF_RECVRY_THREADS and ON_RECVRY_THREADS parameters specify the number of recovery threads, respectively, that operate when Dynamic Server performs a cold restore or fast recovery. The number of threads should typically match the number of tables or fragments that are frequently updated to roll forward the transactions recorded in the logical log. Another estimate is the number of tables or fragments that experience frequent updates. On a single-CPU host, the number of threads should be no fewer than 10 and no more than 30 or 40. At a certain point, the advantages of parallel threads are outweighed by the overhead that is associated with each thread.

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.

DRINTERVAL and DRTIMEOUT

The DRINTERVAL parameter indicates whether the data-replication buffer is flushed synchronously or asynchronously to the secondary database server. If set to flush asynchronously, this parameter specifies the interval between flushes. Each flush impacts the CPU and sends data across the network to the secondary database server.

The DRTIMEOUT parameter specifies the interval for which either database server waits for a transfer acknowledgment from the other. If the primary database server does not receive the expected acknowledgment, it adds the transaction information to the file named in the DRLOSTFOUND configuration parameter. If the secondary database server receives no acknowledgment, it changes data-replication mode as the DRAUTO configuration parameter specifies. For more information on data replication, refer to your *Administrator's Guide*.

ADTERR and ADTMODE

The ADTERR parameter specifies whether Dynamic Server is to halt processing for a user session for which an audit record encounters an error. When ADTERR is set to halt such a session, the response time for that session appears to degrade until one of the successive attempts to write the audit record succeeds. The ADTMODE parameter enables or disables auditing according to the audit records that are specified with the **onaudit** utility. Records are written to files in the directory that the AUDITPATH parameter specifies. The AUDITSIZE parameter specifies the size of each audit record file.

Chapter

4

Table and Index Performance Considerations

This chapter describes performance considerations associated with unfragmented tables, table fragments, and indexes. It discusses the following issues:

- Table placement on disk to increase throughput and reduce contention
- Layout of tables and indexes for more efficient access
- Changing tables to add or delete historical data
- Denormalization of the database to reduce overhead

Placement of Tables on Disk

Tables that Dynamic Server supports reside on one or more portions of a disk or disks. You control the placement of a table on disk when you create it by assigning it to a dbspace. A dbspace is composed of one or more chunks. Each chunk corresponds to all or part of a disk partition. When you assign chunks to dbspaces, you make the disk space in those chunks available for storing tables or table fragments with Dynamic Server.

When you configure chunks and allocate them to dbspaces, you must relate the size of your dbspaces to the tables or fragments that each dbspace is to contain. To estimate the size of a table, follow the instructions in "Estimating Table and Index Size" on page 4-8.

The database administrator (DBA) who is responsible for creating a table assigns that table to a dbspace in one of the following ways:

- The DBA uses the IN DBSPACE clause of the CREATE TABLE statement.
- The DBA uses the dbspace of the current database. The current database is set by the most-recent DATABASE or CONNECT statement that the DBA issued before issuing the CREATE TABLE statement.

The DBA can fragment a table across multiple dbspaces, as described in "Planning a Fragmentation Strategy" on page 6-4, or use the ALTER FRAGMENT statement to move a table to another dbspace. The ALTER FRAGMENT statement provides the simplest method for altering the placement of a table. However, the table is unavailable while Dynamic Server processes the alteration. Schedule the movement of a table or fragment at a time that affects the fewest users. For a description of the ALTER FRAGMENT statement, see the *Informix Guide to SQL: Syntax*.

Other methods exist for moving tables between dbspaces. A DBA can unload the data from a table and then move that data to another dbspace with the LOAD and UNLOAD SQL statements, as the *Informix Guide to SQL: Syntax* describes. The Dynamic Server administrator can perform the same actions with the **onload** and **onunload** utilities, as your *Administrator's Guide* describes.

Moving tables between databases with LOAD and UNLOAD or **onload** and **onunload** involves periods in which data from the table is copied to tape and then reloaded onto the system. These periods present windows of vulnerability during which a table can become inconsistent with the rest of the database. To prevent the table from becoming inconsistent, you must restrict access to the version that remains on disk while the data transfers occur.

Depending on the size, fragmentation strategy, and indexes that are associated with a table, it can be faster to unload a table and reload it than to alter fragmentation. For other tables, it can be faster to alter fragmentation. You might have to experiment to determine which method is faster for a table that you want to move or repartition.

Isolating High-Use Tables

You can place a table with high I/O activity on a dedicated disk device and thus 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-access tables on separate disk devices 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-access table on its own disk device, assign the device to a chunk and assign that chunk to a dbspace, and then place the table in the dbspace that you created. Figure 4-1 shows three high-use tables placed on three disks.

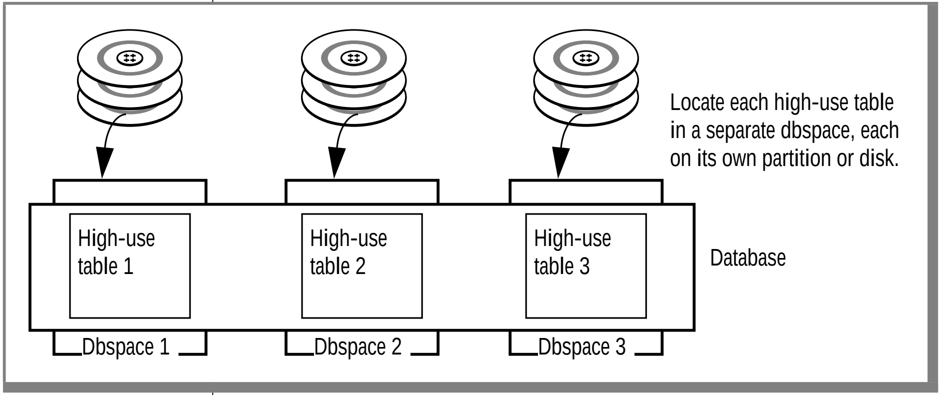

Locate each high-use table in a separate dbspace, each on its own partition or disk.
High-use table 1
High-use table 2
High-use table 3
Database
Dbspace 1
Dbspace 2
Dbspace 3

Figure 4-1
Isolating High-Use Tables

Placing High-Use Tables on Middle Partitions of Disks

To minimize disk-head movement, place the most-frequently accessed data on partitions close to the middle of the disk, as Figure 4-2 shows. This approach minimizes disk-head movement to reach data in the high-demand table.

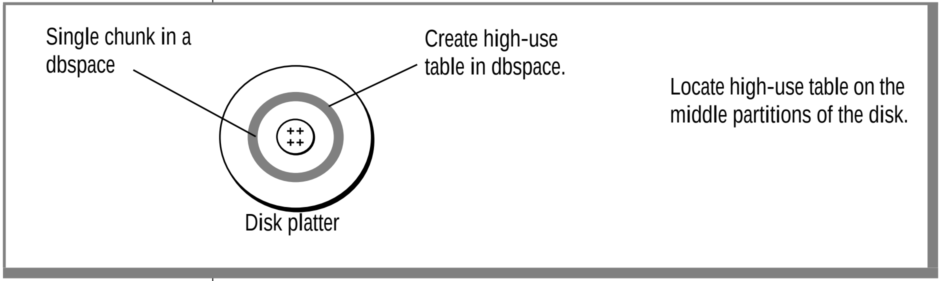

Single chunk in a dbspace
Create high-use table in dbspace.
Locate high-use table on the middle partitions of the disk.
Disk platter

Figure 4-2
Disk Platter with High-Use Table Located on Middle Partitions

To place high-use tables on the middle partition of the disk, create a raw device composed of cylinders that reside midway between the spindle and the outer edge of the disk. (For instructions on how to create a raw device, See your operating-system administrator's guide.) Allocate a chunk, associating it with this raw device, as your *Administrator's Guide* describes. Then create a dbspace with this same chunk as the initial and only chunk. When you create a high-use table, place the table in this dbspace.

Using Multiple Disks for a Dbspace

A dbspace can comprise multiple chunks, and each chunk can represent a different disk. This arrangement allows you to distribute data in a dbspace over multiple disks. Figure 4-3 shows a dbspace distributed over multiple disks.

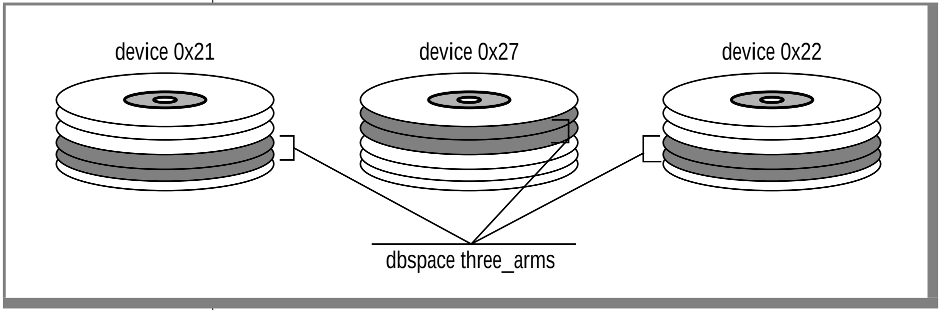


Figure 4-3
A Dbspace Distributed over Three Disks

Using multiple disks for a dbspace helps to distribute I/O across dbspaces that contain several small tables. Because you cannot use this type of distributed dbspace for PDQ, Informix recommends that you use the table-fragmentation techniques described in "Designing a Distribution Scheme" on page 6-12 to partition large, high-use tables across multiple dbspaces.

Spreading Temporary Tables and Sort Files Across Multiple Disks

To define several dbspaces for temporary tables and sort files, use **onspaces -t**. By placing these dbspaces on different disks and listing them in the DBSPACETEMP configuration parameter, you can spread the I/O associated with temporary tables and sort files across multiple disks, as Figure 4-4 illustrates. You can list dbspaces that contain regular tables in DBSPACETEMP.

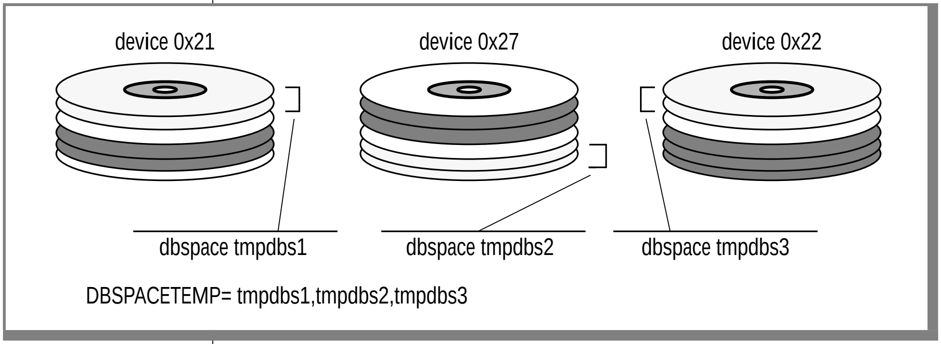


Figure 4-4
Dbspaces for Temporary Tables and Sort Files

Users can specify their own lists of dbspaces for temporary tables and sort files with the **DBSPACETEMP** environment variable. For details, refer to "Parameters and Variables That Affect Temporary Tables and Sorting" on page 3-51.

Backup-and-Restore Considerations

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 rendered 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 in a particular dbspace.

Although you must perform a cold restore if a dbspace that contains critical data fails, you need to perform a warm restore only if a noncritical dbspace fails. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical data. For more information, see your *Archive and Backup Guide* or your *Backup and Restore Guide*.

Improving Performance for Nonfragmented Tables and Table Fragments

The following factors affect the performance of an individual table or table fragment:

- The placement of the table or fragment, as previous sections describe
- The size of the table or fragment
- The indexing strategy used
- The size and placement of table extents with respect to one another
- The frequency access rate to the table

Estimating Table and Index Size

This section discusses methods for calculating the approximate sizes (in disk pages) of tables and indexes.

The disk pages allocated to a table are collectively referred to as a *tblspace*. The tblspace includes data pages and index pages. If there is BYTE or TEXT data associated with the table that is not stored in an alternative dbspace, pages that hold BYTE or TEXT data are also included in the tblspace.

The tblspace does not correspond to any fixed region within a dbspace. The data extents and indexes that make up a table can be scattered throughout the dbspace.

The size of a table includes all the pages within the tblspace: data pages, index pages, and pages that store BYTE or TEXT data. Blobpages that are stored in a separate blobspace or on an optical subsystem are not included in the tblspace and are not counted as part of the table size. The following sections describe how to estimate the page count for each type of page within the tblspace.

Tip: *If an appropriate sample table already exists, or if you can build a sample table of realistic size with simulated data, you do not have to make estimates. You can run* ***oncheck -pt*** *to obtain exact numbers.*

Estimating Data Pages

How you estimate the data pages of a table depends on whether that table contains fixed- or variable-length rows.

Estimating Tables with Fixed-Length Rows

Perform the following steps to estimate the size (in pages) of 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. Use **oncheck -pr** to obtain the size of a page.
2. Subtract 28 from this aFmount to account for the header that appears on each data page. The resulting amount is referred to as *pageuse*.
3. 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 your table, you can using the following SQL statement to obtain the size of a row:

   ```
   SELECT rowsize FROM systables WHERE tabname = 'table-name';
   ```
4. Estimate the number of rows the table is expected to contain. This number is referred to as *rows*.

   The procedure for calculating the number of data pages that a table requires differs depending on whether the row size is less than or greater than *pageuse*.

Important: *Although the maximum size of a row that Dynamic Server accepts is approximately 32 kilobytes, performance degrades 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 4-49.*

5. 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 are to 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.

6. If the size of the row is greater than *pageuse*, Dynamic Server divides the row between pages. The page that contains the initial portion of a row is called the *home page*. Pages that contains 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 calculate the size of the row remainder with the following formula:

      ```
      remsize = rowsize - (pageuse + 8)
      ```

      If *remsize* is less than *pageuse* - 4, you have no full remainder pages. Otherwise, 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 you have accounted for the home and full remainder pages for an individual row. In the following formula, the **remainder()** function notation indicates that you are to take the remainder after division:

```
partremsize = remainder(rowsize/(pageuse - 8)) + 4
```

Dynamic Server uses certain size thresholds with respect to the page size to determine how many partial remainder pages to use. Use the following formula to calculate the ratio of the partial remainder to the page:

```
partratio = partremsize/pageuse
```

Use the appropriate formula in the following table to calculate the number of partial remainder pages.

partratio Value	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. Add up the total number of pages with the following formula:

```
tablesize = homepages + fullrempages + partrempages
```

Estimating Tables with Variable-Length Rows

When a table contains one or more VARCHAR or NVARCHAR columns, its rows can have varying lengths. These varying lengths introduce uncertainty into the calculations. You must form an estimate of 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 Dynamic Server allocates space to rows of varying size, it considers a page to be full when no room exists for an additional row of the maximum size.*

To estimate the size of a table with variable-length rows, you must make the following estimates and choose a value between them, based on your understanding of the data:

- The maximum size of the table, which you calculate based on the maximum width allowed for all VARCHAR or NVARCHAR columns
- The projected size of the table, which you calculate based on a typical width for each VARCHAR or NVARCHAR 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 Tables with Fixed-Length Rows" on page 4-9. 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 within 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 to use some fractional portion of the maximum width, such as 2/3 (.67).
2. Use this value for *rowsize* and perform the calculations described in "Estimating Tables with Fixed-Length Rows" on page 4-9. 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 within that range that seems most reasonable to you. The less familiar you are with the data, the more conservative (higher) your estimate should be.

Estimating Index Pages

The index pages associated with a table can add significantly to the size of a tblspace. As Figure 4-5 shows, an index is arranged as a hierarchy of pages (technically, a *B+ tree*). The topmost 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.

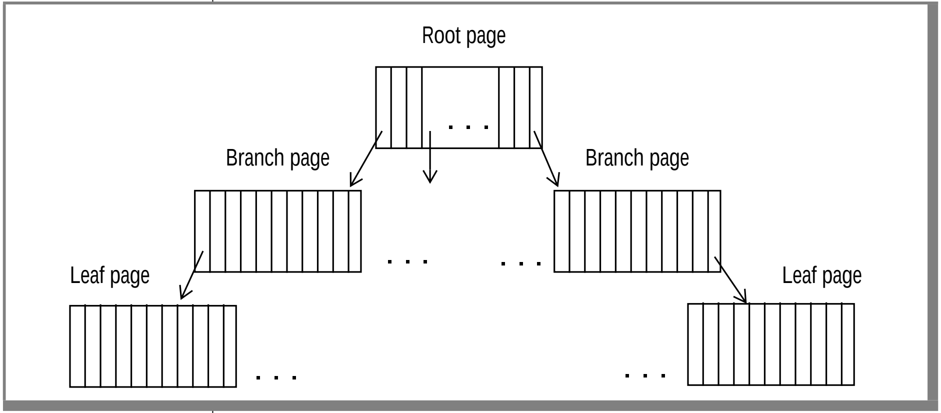


Figure 4-5
B-Tree Structure of an Index

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.

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.

Index entries contained within leaf pages are sorted in key-value order. An index entry consists of a *key*, one or more *row pointers*, a *delete flag*, and, for indexes on fragmented tables, a *fragment ID*. The key is a copy of the indexed columns from one row of data. A row pointer provides an address used to locate a row that contains the key. (A unique index contains one index entry for every row in the table.) The delete flag indicates whether a row with that particular key has been deleted. The fragment ID identifies the table fragment (also referred to as a *partition*) in which the row resides.

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*. Add 4 to *colsize* to obtain *keysize*, the actual size of a key in the index.
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 there are very few duplicate values, use `1` for *propunique*. If there is a significant proportion of duplicate entries, 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 pagesize minus the page header (2,020 for a 2-kilobyte pagesize).

   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 called the *branchtotal*.

10. Use the following formula to calculate the number of pages in the compact index:

    ```
    compactpages = (leaves + branchtotal)
    ```

11. If your Dynamic Server instance uses a fill factor for indexes, the size of the index increases. The fill factor for indexes is set with FILLFACTOR, a Dynamic Server configuration parameter. You can also set the fill factor for an individual index with the FILLFACTOR clause of the CREATE INDEX statement in SQL.

    To incorporate the fill factor into your estimate for index pages, use the following formula:

    ```
    indexpages = 100 * compactpages / FILLFACTOR
    ```

As rows are deleted and new ones are inserted, the number of index entries can vary within a page. 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 **oncheck** utility.

Estimating Tblspace Pages Occupied by BYTE or TEXT Data

In your estimate of the space required for a table, you must include space for BYTE or TEXT data that is to be stored in that tblspace.

To estimate the number of blobpages

1. Obtain the page size with **onstat -c**. Calculate the usable portion of the blobpage with the following formula:

   ```
   bpuse = pagesize - 32
   ```

2. For each byte of blobsize *n*, calculate the number of pages that the byte occupies (*bpages_n*) with the following formula:

   ```
   bpages1 = ceiling(bytesize1/bpuse)
   bpages2 = ceiling(bytesize2/bpuse)
   .
   .
   .
   bpages_n = ceiling(bytesize_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 BYTE or TEXT data, as follows:

   ```
   blobpages = bpages1 + bpages2 + ... + bpages_n
   ```

Alternatively, you can base your estimate on the median size of BYTE or TEXT data; that is, the BYTE or TEXT data size that occurs most frequently. This method is less precise, but it is easier to calculate.

To estimate the number of blobpages based on median BYTE or TEXT data size

1. Calculate the number of pages required for BYTE or TEXT data of median size, as follows:

   ```
   mpages = ceiling(mblobsize/bpuse)
   ```

2. Multiply this amount by the total number of BYTE or TEXT data, as follows:

   ```
   blobpages = blobcount * mpages
   ```

Locating TEXT or BYTE Data in the Tblspace or a Separate Blobspace

When you create a TEXT or BYTE column on magnetic disk, you have the option of locating the column data in the tblspace or in a separate blobspace. You can often improve performance by storing BYTE or TEXT data in a separate blobspace. (You can also store BYTE or TEXT data on optical media, but this discussion does not apply to BYTE or TEXT data that is stored in this way.) In the following example, a TEXT value is located in the tblspace, and a BYTE value is located in a blobspace named **rasters**:

```
CREATE TABLE examptab
    (
    pic_id SERIAL,
    pic_desc TEXT IN TABLE,
    pic_raster BYTE IN rasters
    )
```

A TEXT or BYTE value is always stored apart from the rows of the table; only a 56-byte descriptor is stored with the row. However, the BYTE or TEXT data occupies at least one disk page. The BYTE or TEXT data to which the descriptor points can reside in the same set of extents on disk as the table rows (in the same tblspace) or in a separate blobspace.

When BYTE or TEXT data values are stored in the tblspace, the pages of their data are interspersed among the pages that contain rows, which can greatly increase the size of the table. When the database server reads only the rows and not the BYTE or TEXT data, the disk arm must move farther than when the blobpages are stored apart. The database server scans only the row pages in the following situations:

- Any SELECT operation that does not retrieve a BYTE or TEXT column
- Whenever it tests rows using a filter expression

Another consideration is that disk I/O to and from a dbspace is buffered in shared memory of the database server. Pages are stored in case they are needed again soon, and when pages are written, the requesting program can continue before the actual disk write takes place. However, because blobspace data is expected to be voluminous, disk I/O to and from blobspaces is not buffered, and the requesting program is not allowed to proceed until all output has been written to the blobspace.

For best performance, locate a TEXT or BYTE column in a blobspace in either of the following circumstances:

- When single data items are larger than one or two pages each
- When the number of pages of BYTE or TEXT data is more than half the number of pages of row data.

For a table that is both relatively small and nonvolatile, you can achieve the effect of a dedicated blobspace by separating row pages and blobpages, as the following paragraphs explain.

To separate row pages from blobpages

1. Load the entire table with rows in which the BYTE or TEXT data columns are null.
2. Create all indexes.
3. The row pages and the index pages are now contiguous.
4. Update all the rows to install the BYTE or TEXT data.

   The blobpages now appear after the pages of row and index data within the tblspace.

Managing Indexes

An index is necessary on any column or composition of columns that must be unique. However, as discussed in Chapter 7, "Queries and the Query Optimizer," the presence of an index can also allow the query optimizer to speed up a query. The optimizer can use an index in the following ways:

- To replace repeated sequential scans of a table with nonsequential access
- To avoid reading row data when processing expressions that name only indexed columns
- To avoid a sort (including building a temporary table) when executing the GROUP BY and ORDER BY clauses

As a result, an index on the appropriate column can save thousands, tens of thousands, or in extreme cases, even millions of disk operations during a query. However, indexes entail costs.

Space Costs of Indexes

The first cost of an index is disk space. An estimating method appears in "Estimating Index Pages" on page 4-13. The presence of an index can add many pages to a tblspace; it is easy to have as many index pages as row pages in an indexed table.

Time Costs of Indexes

The second cost of an index is time whenever the table is modified. The following descriptions assume that approximately two pages must be read to locate an index entry. That 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 presumed 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 referencing such an index.

Presumably, one index is used to locate a row being altered. The pages for that index might be found in page buffers in Dynamic Server shared memory; however, the pages for any other indexes that need altering must be read from disk.

Under these assumptions, index maintenance adds time to different kinds of modifications, as the following list shows:

- When you delete a row from a table, its entries must be deleted from all indexes.

  The entry for the deleted row must be looked up (two or three pages in), and the leaf page must be rewritten. 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 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), and then the new column value must be located in the same index (two or three more pages in) and the row entered (one more deferred page out).

Insertions and deletions change the number of entries on a leaf page. Although virtually every *pagents* operation requires some additional work to deal with a leaf page that has either filled up or been emptied, if *pagents* is greater than 100, this additional work occurs less than 1 percent of the time, and you can often disregard it when you estimate the I/O impact.

In short, when a row is inserted or deleted at random, allow three to four added page I/O operations per 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, it is clear that index maintenance is the most time-consuming part of data modification. One way to reduce this cost is discussed in "Clustering" on page 4-23.

Choosing Columns for Indexes

Indexes are required on columns that must be unique and are not specified as primary keys. In addition, add an index on columns that:

- are used in joins that are not specified as foreign keys.
- are frequently used in filter expressions.
- are frequently used for ordering or grouping.
- do not involve duplicate keys.
- are amenable to clustered indexing.

Filtered Columns in Large Tables

If a column is often used to filter the rows of a large table, consider placing an index on it. The optimizer can use the index to select the desired columns 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 a subset of rows, consider putting an index on that column.

This strategy yields a net savings of time only when the selectivity of the column is high; that is, when only a small fraction of rows holds any one indexed value. Nonsequential access through an index takes several more disk I/O operations than sequential access does, so if a filter expression on the column passes more than a fourth 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.
- The column contains at least 100 unique values.
- Most column values appear in fewer than 10 percent of the rows.

Order-By and Group-By Columns

When a large quantity of 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. But, as discussed in Chapter 7, if the ordering columns are indexed, the optimizer sometimes reads the rows in sorted order through the index, thus avoiding a final sort.

Because the keys in an index are in sorted sequence, the index really represents the result of sorting the table. By placing an index on the ordering column or columns, you can replace many sorts during queries with a single sort when the index is created.

Avoiding Columns with Duplicate Keys

When duplicate keys are permitted in an 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 quite long and, in fact, can cross multiple leaf pages.

Placing an index on a column that has low selectivity (that is, a small number of distinct values relative to the number of rows) can reduce performance. In such cases, the database server must not only search the entire set of rows that match the key value, but it must also lock all the affected data and index pages. This process can impede the performance of other update requests as well.

To correct this problem, replace the index on the low-selectivity column with a composite index that has a higher selectivity. Use the low-selectivity column as the leading column and a high-selectivity column as your second column in the index. The composite index limits the number of rows that the database server must search to locate and apply an update.

You can use any second column to disperse the key values as long as its value does not change or changes at the same time as the real key. The shorter the second column the better, because its values are copied into the index and expand its size.

Clustering

Clustering is a method for arranging the rows of a table so that their physical order on disk closely corresponds to the sequence of entries in the index. (Do not confuse the clustered index with an *optical cluster*, which is a method for storing logically related BYTE or TEXT data together on an optical volume.)

When you know that a table is ordered by a certain index, you can avoid sorting. You can also be sure that when the table is searched on that column, it is read effectively in sequential order instead of nonsequentially. These points are covered in Chapter 7.

In the **stores7** database, the **orders** table has an index, **zip_ix,** on the postal-code column. The following statement causes the database server to put the rows of the **customer** table into descending order by postal code:

```
ALTER INDEX zip_ix TO CLUSTER
```

To cluster a table on a nonindexed column, you must create an index. The following statement reorders the **orders** table by order date:

```
CREATE CLUSTERED INDEX o_date_ix ON orders (order_date ASC)
```

To reorder a table, the database server must copy the table. In the preceding example, the database server reads all the rows in the table and constructs an index. Then it reads the index entries in sequence. For each entry, it reads the matching row of the table and copies it to a new table. The rows of the new table are in the desired sequence. This new table replaces the old table.

Clustering is not preserved when you alter a table. When you insert new rows, they are stored physically at the end of the table regardless of their contents. 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 can be restored after the order of rows is disturbed by ongoing updates. The following statement reorders the table to restore data rows to the index sequence:

```
ALTER INDEX o_date_ix TO CLUSTER
```

Reclustering is usually quicker than the original clustering because reading out the rows of a nearly clustered table has an I/O impact that is similar to a sequential scan.

Clustering and reclustering take a lot of space and time. You can avoid some clustering by initially building the table in the desired order.

Dropping Indexes

In some applications, most table updates can be confined to a single time period. You can 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 two good effects.

First, with fewer indexes to update, the updating program can run faster. 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. (The time cost of updating indexes is discussed in "Time Costs of Indexes" on page 4-19.)

Second, newly made indexes are the most efficient ones. Frequent updates tend to dilute the index structure, causing it to contain many partly full leaf pages. This dilution reduces the effectiveness of an index and wastes disk space.

As another time-saving measure, make sure that a batch-updating program calls for rows in the sequence that the primary-key index defines. That sequence causes the pages of the primary-key index to be read in order and only one time each.

The presence of indexes also slows down the population of tables when you use the LOAD statement or the **dbload** utility. Loading a table that has no indexes is a quick process (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.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. It saves 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.

Checking Indexes

You can improve concurrency on a table when you run the **oncheck** utility to check the consistency of index pages or to print the index pages. When you specify the **-w** option with the following options, the **oncheck** utility does not place an exclusive lock on the table. The database server places a shared lock on the table during the execution of the following **oncheck** options:

- **-ciw**
- **-cIw**
- **-pkw**
- **-pKw**
- **-plw**
- **-pLw**

When you use the **-w** option, the **oncheck** utility places an *intent shared* (IS) lock on the table. An IS lock allows other users to read and update the table during the execution of the **oncheck** utility on an index. The IS lock also prevents DDL activity (such as a DROP INDEX or DROP TABLE statements) during the execution of **oncheck**. For more information on lock types, refer to “Monitoring and Administering Locks” on page 5-13.

If you do not use the **-w** option, the table can be unavailable to other users for DML activity (UPDATE, INSERT, or DELETE statements) for a significant amount of time when you execute the **oncheck** utility on a very large table.

To improve concurrency on a table when you run the oncheck utility on an index

1. Ensure that the table uses row locking.

   If the table uses page locking, the database server returns the following error message when you run **oncheck** utility with the **-w** option:

   ```
   WARNING: index check requires a s-lock on tables whose
   lock level is page.
   ```

   You can query the **systables** system catalog to see the current lock level of the table, as the following sample SQL statement shows:

   ```
   SELECT locklevel FROM systables
      WHERE tabname = "customer"
   ```

   If you do not see a value of `R` (for row) in the **locklevel** column, you can modify the lock level, as the following sample SQL statement shows:

   ```
   ALTER TABLE tab1 LOCK MODE (ROW);
   ```

2. Run the **oncheck** utility with the **-w** option to place a shared lock on the table.

   The following sample **oncheck** commands show index options with the **-w** option:

   ```
   oncheck -ciw stores7:informix.customer
   oncheck -pLw stores7:informix.customer
   ```

If a user updates, inserts, or deletes column values that are key values in the index that **oncheck** is checking, the user's changes still take effect**.** If **oncheck** encounters an index page that the user updated, it ignores the updated page. If you want 100-percent assurance that the index is good, do not use the **-w** option of **oncheck.** When you do not use the **-w** option, **oncheck** places an exclusive lock on the table.

Improving Performance for Index Builds

Whenever possible, the database server uses parallel processing to improve the response time of index builds. The number of parallel processes is based on the number of fragments in the index and the value of the **PSORT_NPROCS** environment variable. The database server builds the index with parallel processing even when the value of PDQ priority is `0`.

You can often improve the performance of an index build by taking the following steps:

1. Set PDQ priority to a value greater than 0 to obtain more memory than the default 128 kilobytes.

   When you set PDQ priority to greater than 0, the index build can take advantage of the additional memory for parallel processing.

   To set PDQ priority, use either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement in SQL.

2. Do not set the **PSORT_NPROCS** environment variable.

   Informix recommends that you not set the **PSORT_NPROCS** environment variable. If you have a computer with multiple CPUs, the database server uses two threads per sort when it sorts index keys and **PSORT_NPROCS** is not set. The number of sorts depends on the number of fragments in the index, the number of keys, the key size, and the values of the PDQ memory configuration parameters.

3. Allocate enough memory and temporary space to build the entire index.

   a. Estimate the amount of virtual shared memory that the database server might need for sorting.

      For more information, refer to "Estimating Sort Memory" on page 4-28.

   b. Specify more memory with the DS_TOTAL_MEMORY and DS_MAX_QUERIES configuration parameters.

   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 4-29.

   d. Use the **onspaces -t** utility to create large temporary dbspaces and specify them in the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

      For information on how to optimize temporary dbspaces, refer to "Dbspaces for Temporary Tables and Sort Files" on page 3-50.

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 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 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 PDQ priority is 0, the maximum amount of shared memory that the database server allocates for a sort is about 128 kilobytes.

If PDQ priority is greater than 0, the maximum amount of shared memory that the database server allocates for a sort is controlled by the memory grant manager (MGM). The MGM uses the settings of PDQ priority and the following configuration parameters to determine how much memory to grant for the sort:

- DS_TOTAL_MEMORY
- DS_MAX_QUERIES
- MAX_PDQPRIORITY

For more information about allocating memory for parallel processing, refer to “Allocating Resources for PDQ Queries” on page 9-7.

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:
   a. For a nonfragmented table and a fragmented table with an index created without an explicit fragmentation strategy, use the following formula
      ```
      sizeof_sort_item = keysize + 4
      ```
   b. For fragmented tables with the index explicitly fragmented, use the following formula:
      ```
      sizeof_sort_item = keysize + 8
      ```
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.

Managing Extents

As you add rows to a table, Dynamic Server allocates disk space to it in units called *extents*. Each extent is a block of physically contiguous pages from the dbspace. Even when the dbspace includes more than one chunk, each extent is allocated entirely within a single chunk so that it remains contiguous.

Contiguity is important to performance. When the pages of data are contiguous, disk-arm motion is minimized when the database server reads the rows sequentially. The mechanism of extents is a compromise between the following competing requirements:

- Most dbspaces are shared among several tables.
- The size of some tables is not known in advance.
- Tables can grow at different times and different rates.
- All the pages of a table should be adjacent for best performance.

Because table sizes are not known, table space cannot be preallocated. Therefore, Dynamic Server adds extents only as they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when Dynamic Server creates a new extent that is 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 the extents to be added as the table grows. The following example creates a table with a 512-kilobyte initial extent and 100-kilobyte added 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. For example, if you have a 2-kilobyte page, the default length is 16 kilobytes.

You can change the size of extents to be added with 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 table, you might want to adjust the next-extent size because each fragment requires less space than the original, unfragmented table. If the unfragmented table was defined with a large next-extent size, Dynamic Server uses that same size for the next-extent on *each* fragment, which results in over-allocation of disk space. Each fragment requires only a proportion of the space for the entire table.

For example, if you fragment the preceding **big_one** sample table across five disks, 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 size:

```
ALTER TABLE big_one MODIFY NEXT SIZE 20
```

The next-extent sizes of the following kinds of tables are not very important to performance:

- A small table is defined as a table that has only one extent. If such a table is heavily used, large parts of it remain buffered in memory.
- An infrequently used table is not important to performance no matter what size it is.
- A table that resides in a dedicated dbspace always receives new extents that are adjacent to its old extents. The size of these extents is not important because, being adjacent, they perform as one large extent.

When you assign an extent size to these kinds of tables, the only consideration is to avoid creating large numbers of extents. Having a large number of extents causes the database server to spend extra time finding the data. In addition, an upper limit exists on the number of extents allowed. (This topic is covered in "Upper Limit on Extents" on page 4-32.)

No upper limit exists on extent sizes except the size of the chunk. When you know the final size of a table (or can confidently predict it within 25 percent), allocate all its space in the initial extent. When tables grow steadily to unknown size, assign them next-extent sizes that let them share the dbspace with a small number of extents each. The following steps outline one possible approach.

To allocate space for table extents

1. Decide how to allocate space among the tables. For example, you might divide the dbspace among three tables in the ratio 0.4: 0.2: 0.3 (reserving 10 percent for small tables and overhead).
2. Give each table one-fourth of its share of the dbspace as its initial extent.
3. Assign each table one-eighth of its share as its next-extent size.
4. Monitor the growth of the tables regularly with **oncheck**.

As the dbspace fills up, you might not have enough contiguous space to create an extent of the specified size. In this case, Dynamic Server allocates the largest contiguous extent that it can.

Upper Limit on Extents

Do not allow a table to acquire a large number of extents because an upper limit exists on the number of extents allowed. Trying to add an extent after the limit is reached causes error -136 (`No more extents`) to follow an INSERT request.

The upper limit on extents depends on the page size and the table definition. To learn 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 BYTE or TEXT data 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 **oncheck -pr**.

The table can have no more than *maxextents* extents.

To help ensure that the limit is not exceeded:

- Dynamic Server checks the number of extents each time that it creates a new extent. If the number of the extent being created is a multiple of 16, Dynamic Server automatically doubles the next-extent size for the table. Therefore, at every sixteenth creation, Dynamic Server doubles the next-extent size.
- When Dynamic Server creates a new extent adjacent to the previous extent, it treats both extents as a single extent.

Checking for Extent Interleaving

When two or more growing tables share a dbspace, extents from one tblspace can be placed between extents from another tblspace. When this situation occurs, the extents are said to be *interleaved*. Interleaving creates gaps between the extents of a table, as Figure 4-6 shows. Performance suffers when disk seeks for a table must span more than one extent, particularly for sequential scans. Try to optimize the table-extent sizes, which limits head movement. Also consider placing the tables in separate dbspaces.

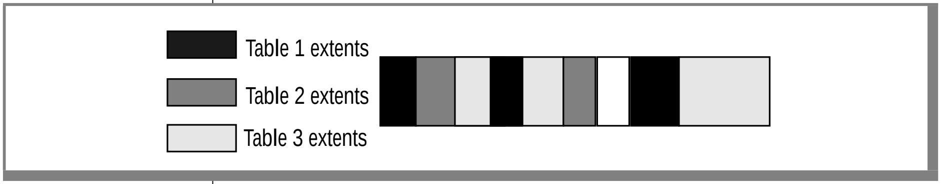


Figure 4-6
Interleaved Table Extents

Periodically check for extent interleaving by monitoring Dynamic Server chunks. Execute **oncheck -pe** 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 Dynamic 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 tables with the UNLOAD and LOAD statements.
- Create or alter an index to cluster.
- Use the ALTER TABLE statement.

Reorganizing Dbspaces and Tables to Eliminate Extent Interleaving

You can rebuild a dbspace to eliminate interleaved extents, as Figure 4-7 illustrates. The order of the reorganized tables within the dbspace is not important. All that matters is that the pages of each reorganized table are together so that no lengthy seeks are required to read the table sequentially. When reading a table nonsequentially, the disk arm ranges only over the space occupied by that table.

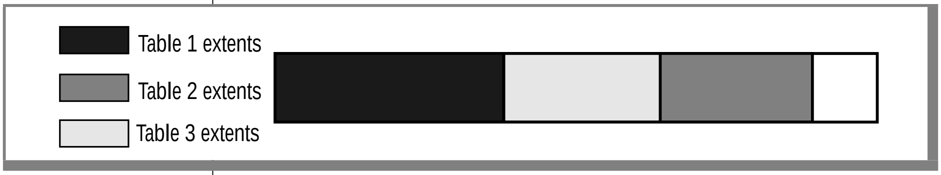


Figure 4-7
A Dbspace Reorganized to Eliminate Interleaved Extents

To reorganize tables in a dbspace

1. Copy the tables in the dbspace to tape individually with the UNLOAD statement in DB-Access.
2. Drop all the tables in the dbspace.
3. Re-create the tables with the LOAD statement or the **dbload** utility.

The LOAD statement re-creates the tables with the same properties as they had before, including the same extent sizes.

You can also unload a table with the **onunload** utility and reload the table with the companion **onload** utility. For further information about selecting the correct utility or statement to use, refer to the *Informix Migration Guide*.

Creating or Altering an Index to Cluster

Depending on the circumstances, you can eliminate extent interleaving if you create a clustered index or alter an index to cluster. When you use the TO CLUSTER clause of the CREATE INDEX or ALTER INDEX statement, Dynamic Server sorts and reconstructs the table. The TO CLUSTER clause reorders rows in the physical table to match the order in the index. For more information, refer to "Clustering" on page 4-23.

The TO CLUSTER clause eliminates interleaved extents under the following conditions:

- The chunk must contain contiguous space that is large enough to rebuild each table.
- Dynamic Server must use this contiguous space to rebuild the table.

  If blocks of free space exist before this larger contiguous space, Dynamic Server might allocate the smaller blocks first. Dynamic Server allocates space for the ALTER 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 Dynamic 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 **oncheck -pe** command.

To use the TO CLUSTER clause of the ALTER INDEX statement

1. For each table in the chunk, drop all fragmented or detached indexes except the one that you want to cluster.
2. Cluster the remaining index with the TO CLUSTER clause of the ALTER INDEX statement.

   This step eliminates interleaving the extents when you rebuild the table by rearranging the rows.
3. Re-create all the other indexes.

   You compact the indexes in this step because Dynamic Server sorts the index values before it adds them to the B-tree.

You do not need to drop an index before you cluster it. However, the ALTER INDEX process is faster than CREATE INDEX because Dynamic Server reads the data rows in cluster order using the index. In addition, the resulting indexes are more compact.

To prevent the problem 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, Dynamic Server copies and reconstructs the table. When Dynamic Server reconstructs the entire table, the table is rewritten onto other areas of the dbspace. However, if other tables are in the dbspace, no guarantee exists that the new extents will be adjacent to each other.

Important: *For certain types of operations that you specify in the ADD, DROP, and MODIFY clauses, Dynamic Server does not copy and reconstruct the table during the ALTER TABLE operation. In these cases, Dynamic Server uses an in-place alter algorithm to modify each row when it is updated (rather than during the ALTER TABLE operation). For more information on the conditions for this in-place alter algorithm, refer to "In-Place Alter" on page 4-41.*

Reclaiming Unused Space Within an Extent

Once Dynamic Server allocates disk space to a tblspace as part of an extent, that space remains dedicated to the tblspace. Even if all extent pages become empty after deleting data, the disk space remains unavailable for use by other tables.

Important: *When you delete rows in a table, Dynamic Server reuses that space to insert new rows into the same table. This section describes procedures to reclaim unused space for use by other tables.*

You might want to resize a table that does not require the entire amount of space that was originally allocated to it. You can reallocate a smaller dbspace and release the unneeded space for other tables to use.

As the Dynamic Server administrator, you can reclaim the disk space in empty extents and make it available to other users by rebuilding the table. To rebuild the table, use any of the following SQL statements:

- ALTER INDEX
- UNLOAD and LOAD
- ALTER FRAGMENT

Reclaiming Space in an Empty Extent with ALTER INDEX

If the table with the empty extents includes an index, you can execute the ALTER INDEX statement with the TO CLUSTER clause. Clustering an index rebuilds the table in a different location within the dbspace. All the extents associated with the previous version of the table are released. Also, the newly built version of the table has no empty extents.

For more information about the syntax of the ALTER INDEX statement, refer to the *Informix Guide to SQL: Syntax*. For more information on clustering, refer to "Clustering" on page 4-23.

Reclaiming Space in an Empty Extent with the UNLOAD and LOAD Statements or the onunload and onload Utilities

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 with the UNLOAD and LOAD statements or the **onunload** and **onload** utilities. For further information about selecting the correct utility or statement to use, refer to the *Informix Migration Guide*.

For more information about the syntax of the UNLOAD and LOAD statements, refer to the *Informix Guide to SQL: Syntax*.

Releasing Space in an Empty Extent with ALTER FRAGMENT

You can use the ALTER FRAGMENT statement to rebuild a table, which releases space within the extents that were allocated to that table. For more information about the syntax of the ALTER FRAGMENT statement, refer to the *Informix Guide to SQL: Syntax*.

Changing Tables

You might want to change an existing table for various reasons:

- To periodically refresh large decision-support tables with data
- To add or drop historical data from a certain 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 tables that have been unloaded from active OLTP databases. Dynamic Server can quickly load large tables by using a series of SQL statements and external tables.

For more information on how the database server performs high performance loading, refer to your *Administrator's Guide*.

Dropping Indexes Before Loading or Updating Tables

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, 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. (The time cost of updating indexes is discussed in "Time Costs of Indexes" on page 4-19.)
- 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 one time each.

The presence of indexes also slows down the population of tables when you use the LOAD statement or the **dbload** utility. Loading a table that has no indexes is a quick process (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.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. 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.

Attaching or Detaching Fragments

Many customers use ALTER FRAGMENT ATTACH and DETACH statements to perform data-warehouse-type operations. 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 into an existing table incrementally by taking advantage of the fragmentation technology.

For more information on how to take advantage of the performance enhancements for the ATTACH and DETACH options of the ALTER FRAGMENT statement, refer to "Improving the Performance of Attaching and Detaching Fragments" on page 6-29.

Altering a Table Definition

The database server uses one of several algorithms to process an ALTER TABLE statement in SQL:

- Slow alter
- In-place alter
- Fast alter

Slow Alter

When the database server uses the slow alter algorithm to process an ALTER TABLE statement, the table can be unavailable to other users for a long period of time because the database server:

- locks the table in exclusive mode for the duration of the ALTER TABLE operation.
- makes a copy of the table to convert the table to the new definition.
- converts the data rows during the ALTER TABLE operation.
- can treat the ALTER TABLE statement as a long transaction and abort it if the LTXHWM threshold is exceeded.

The database server uses the slow alter algorithm when the ALTER TABLE statement makes column changes that cannot be performed in place, such as the following changes:

- Add or drop a column created with the ROWIDS or CRCOLS keyword
- Drop a column of type TEXT or BYTE
- Modify the data type of a column in a way that some possible values of the old type cannot be converted to the new type

  For example, if you modify a column of type INTEGER to CHAR(n), the database server uses the slow alter algorithm if the value of *n* is less than 11. An INTEGER requires 10 characters plus one for the minus sign for the lowest possible negative values.
- Modify the data type of a fragmentation column in a way that value conversion might cause rows to move to another fragment

In-Place Alter

The in-place alter algorithm provides the following performance advantages over the slow alter algorithm:

- Increases table availability

  Other users can access the table 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.

  This increase in table availability can increase system throughput for application systems that require 24 by 7 operations.

  When the database server uses the in-place alter algorithm, the database server locks the table for a shorter time than the slow alter algorithm because the database server:

  - does not make a copy of the table to convert the table to the new definition.
  - does not convert the data rows during the ALTER TABLE operation.
  - alters the physical columns 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.

- Requires less space than the slow alter algorithm

  When the ALTER TABLE operation uses the slow alter algorithm, the database server makes a copy of the table to convert the table to the new definition. The ALTER TABLE operation requires space at least twice the size of the original table plus log space.

  When the ALTER TABLE operation uses the in-place alter algorithm, the space savings can be substantial for very large tables.

- Improves system throughput during the ALTER TABLE operation

  The database server does not need to log any changes to the table data during the in-place alter operation. Not logging changes has the following advantages:

  - Log space savings can be substantial for very large tables.
  - The alter operation is not a long transaction.

When the Database Server Uses the In-Place Alter Algorithm

The database server uses the in-place alter algorithm for certain types of operations that you specify in the ADD, DROP, and MODIFY clauses of the ALTER TABLE statement:

- Add a column or list of columns of any data type except columns that you add with the ROWIDS or CRCOLS keywords
- Drop a column of any data type except type BYTE or TEXT and columns created with the ROWIDS or CRCOLS keywords
- Modify a column for which the database server can convert all possible values of the old data type to the new data type.
- Modify a column that is part of the fragmentation expression if value changes do *not* require a row to move from one fragment to another fragment after conversion.

Figure 4-8 shows the conditions under which the ALTER TABLE MODIFY statement uses the in-place alter algorithm.

Figure 4-8
Operations and Conditions That Use the In-Place Alter Algorithm to Process the ALTER TABLE MODIFY Statement

Operation on Column	Condition
Convert a SMALLINT column to an INTEGER column	All
Convert a SMALLINT column to a DEC(p2,s2) column	p2-s2 >= 5
Convert a SMALLINT column to a DEC(p2) column	p2-s2 >= 5 OR nf
Convert a SMALLINT column to an SMALLFLOAT column	All

Notes:

- The column type DEC(p) refers to non-ANSI databases in which this type is handled as floating point.
- In ANSI databases, DEC(p) defaults to DEC(p,0) and uses the same alter algorithm as DEC(p,s).
- The condition "nf" is when the modified column is not part of the table fragmentation expression.
- The condition indicates that the database server uses the in-place alter algorithm for all cases of the specific column operation.

Operation on Column	Condition
Convert a SMALLINT column to an FLOAT column	All
Convert a SMALLINT column to a CHAR(n) column	n >= 6 AND nf
Convert an INT column to a DEC(p2,s2) column	p2-s2 >= 10
Convert an INT column to a DEC(p2) column	p2 >= 10 OR nf
Convert a INT column to an SMALLFLOAT column	nf
Convert a INT column to an FLOAT column	All
Convert a INT column to a CHAR(n) column	n >= 11 AND nf
Convert a DEC(p1,s1) column to a SMALLINT column	p1-s1 < 5 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to an INTEGER column	p1-s1 < 10 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a DEC(p2,s2) column	p2-s2 >= p1-s1 AND (s2 >= s1 OR nf)
Convert a DEC(p1,s1) column to a DEC(p2) column	p2 >= p1 OR nf
Convert a DEC(p1,s1) column to a SMALLFLOAT column	nf
Convert a DEC(p1,s1) column to a FLOAT column	nf
Convert a DEC(p1,s1) column to a CHAR(n) column	n >= 8 AND nf
Convert a DEC(p1) column to a DEC(p2) column	p2 >= p1 OR nf
Convert a DEC(p1) column to a SMALLFLOAT column	nf

Notes:

- The column type DEC(p) refers to non-ANSI databases in which this type is handled as floating point.
- In ANSI databases, DEC(p) defaults to DEC(p,0) and uses the same alter algorithm as DEC(p,s).
- The condition "nf" is when the modified column is not part of the table fragmentation expression.
- The condition indicates that the database server uses the in-place alter algorithm for all cases of the specific column operation.

Operation on Column	Condition
Convert a DEC(p1) column to a FLOAT column	nf
Convert a DEC(p1) column to a CHAR(n) column	n >= 8 AND nf
Convert a SMALLFLOAT column to a DEC(p2) column	nf
Convert a SMALLFLOAT column to a FLOAT column	nf
Convert a SMALLFLOAT column to a CHAR(n) column	n >= 8 AND nf
Convert a FLOAT column to a DEC(p2) column	nf
Convert a FLOAT column to a SMALLFLOAT column	nf
Convert a FLOAT column to a CHAR(n) column	n >= 8 AND nf
Convert a CHAR(m) column to a CHAR(n) column	n >= m OR (nf AND not ANSI mode)
Increase the length of a CHARACTER column	Not ANSI mode
Increase the length of a DECIMAL or MONEY column	All

Notes:

- The column type DEC(p) refers to non-ANSI databases in which this type is handled as floating point.
- In ANSI databases, DEC(p) defaults to DEC(p,0) and uses the same alter algorithm as DEC(p,s).
- The condition "nf" is when the modified column is not part of the table fragmentation expression.
- The condition indicates that the database server uses the in-place alter algorithm for all cases of the specific column operation.

Performance Considerations for DML Statements

Each time you execute an ALTER TABLE statement that uses the in-place alter algorithm, the database server creates a new version of the table structure. The database server keeps track of all versions of table definitions. The database server resets the version status, all of the version structures and alter structures, until the entire table is converted to the final format or a slow alter is performed.

If the database server detects any down-level version page during the execution of DML statements (INSERT, UPDATE, DELETE, SELECT), it performs the following actions:

- For UPDATE statements, the database server converts the entire data page or pages to the final format.
- For INSERT statements, the database server converts the inserted row to the final format and inserts it into the best fit page. The database server converts the existing rows on the best-fit 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 data pages 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.

Performance Considerations for DDL Statements

The **oncheck -pT *tablename*** option displays data-page versions for outstanding in-place alter operations. An in-place alter is outstanding when data pages still exist with the old definition.

Figure 4-9 displays a portion of the output that the following **oncheck** command produces after four in-place alter operations are executed on the customer demonstration table:

```
oncheck -pT stores7:customer
```

```
...
Home Data Page Version Summary

                Version                        Count

                0 (oldest)                       2
                1                                0
                2                                0
                3                                0
                4 (current)                      0
...
```

Figure 4-9
Sample oncheck -pT Output for Customer Table

The **Count** field in Figure 4-9 displays the number of pages that currently use that version of the table definition. This **oncheck** output shows that four versions are outstanding:

- A value of 2 in **Count** field for the oldest version indicates that two pages use the oldest version.
- A value of 0 in the **Count** fields for the next four versions indicate that no pages have been converted to the latest table definition.

Important: *As you perform more in-place alters on a table, each subsequent ALTER TABLE statement takes more time to execute than the previous statement. Therefore, Informix recommends that you do not have more than approximately 50 to 60 outstanding alters on a table. A large number of outstanding alters affects only the subsequent ALTER TABLE statements. The performance of SELECT statements does not degrade because of the large number of outstanding alters.*

You can convert 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, the **oncheck -pT** command displays the total number of data pages in the **Count** field for the current version of the table.

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 more than one algorithm is in use

  If the ALTER TABLE statement contains more than one change, the database server uses the algorithm with the lowest performance in the execution of the statement.

  For example, assume that an ALTER TABLE MODIFY statement converts a SMALLINT column to a DEC(8,2) column and converts an INTEGER column to a CHAR(8) column. The conversion of the first column is an in-place alter operation, but the conversion of the second column is a slow alter operation. The database server uses the slow alter algorithm to execute this statement.

- When values have to move to another fragment

  For example, suppose you have a table with two integer columns and the following fragment expression:

  ```
  col1 < col2 in dbspace1, remainder in dbspace2
  ```

  If you execute the following ALTER TABLE MODIFY statement, the database server stores a row (4, 30) in **dbspace1** before the alter but stores it in **dbspace2** after the alter operation because 4 < 30 but "30" < "4".

Altering a Column That Is Part of an Index

If the altered column is part of an index, the table is still altered in place, although in this case the server rebuilds the index or indexes implicitly. If you do not need to have the index rebuilt, you should drop or disable the index before you perform the alter operation. Taking these steps improves performance.

However, if the column that you modify is a primary key or foreign key and you want to keep this constraint, you must respecify those keywords in the ALTER TABLE statement, and the database server rebuilds the index.

For example, suppose 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. The in-place alter algorithm provides the following performance benefits:

- Does not make a copy of the table 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.*

Fast Alter

The database server uses the fast alter algorithm when the ALTER TABLE statement changes attributes of the table but does not affect the data. The database server uses the fast alter algorithm when you use the ALTER TABLE statement to:

- change the next-extent size.
- add or drop a constraint.
- change the lock mode of the table.
- change the unique index attribute without modifying the column type.

When the database server uses the fast alter algorithm, the database server holds the lock on the table for a very short time. In some cases, the database server only locks the system catalog tables to change the attribute. In either case, the table is unavailable for queries for only a very short period of time.

Denormalizing the Data Model to Improve Performance

The entity-relationship data model that the *Informix Guide to SQL: Tutorial* describes produces tables that contain no redundant or derived data. According to the tenets of relational theory, these tables are well structured.

Sometimes, to meet extraordinary demands for high performance, you might have to modify the data model in ways that are undesirable from a theoretical standpoint. This section describes some modifications and their associated costs.

Shorter Rows for Faster Queries

Usually, 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 a nonsequential access can be performed from a buffer.

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 rows of awkward lengths. You can shorten the rows by breaking out columns into separate tables that are associated by duplicate key values in each table. As the rows get shorter, query performance should improve.

Expelling Long Strings

The most bulky 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:

- Use VARCHAR columns.
- Use TEXT data.
- Move strings to a companion table.
- Build a symbol table.

Using VARCHAR Strings

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 other 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 Data

When a string fills half a disk page or more, consider converting it to a TEXT column in a separate blobspace. The column within the row page is only 56 bytes long, which allows more rows on a page than when you include a long string. However, the TEXT data type is not automatically compatible with existing programs. The application needed to fetch a TEXT value is a bit more complicated than the code for fetching a CHAR value into a program.

Moving Strings to a Companion Table

Strings that are less than half a page waste disk space if you treat them as TEXT data, but you can remove them from the main table to a companion table.

Building a Symbol Table

If a column contains strings that are not unique in each row, you can remove those strings to a table in which only unique copies are stored.

For example, 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 shown in the following example:

```
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. It is not a logical error; 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*.

Besides changing programs that insert data, you also must 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. Split the table into two tables, a primary table and a companion table, repeating the primary key in each one. The shorter rows allow each table to be queried or updated quickly.

Division 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 demonstration database, 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.

Division by Frequency of Use

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, they can be moved to a companion table. In the demonstration database, it might be that the **ship_instruct**, **ship_weight**, and **ship_charge** columns are queried only in one program. In that case, you can move them to a companion table.

Division by Frequency of Update

Updates take longer than queries, and updating programs lock index pages and rows of data during the update process, preventing 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.

Costs of Companion Tables

Splitting a 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. You can use the methods described in earlier sections to estimate the number of added pages.

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.

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 within a single transaction, for example), you lose semantic integrity.

Redundant Data

Normalized tables contain no redundant 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 makes updating the tables as easy as possible. However, normalized tables can force you to use joins and aggregate functions often, and that can be time consuming.

As an alternative, you can introduce new columns that contain redundant data, provided you understand the trade-offs involved.

Adding Redundant Data

A correct data model avoids redundancy by keeping any attribute only in the table for the entity that it describes. 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.

In the **stores7** sample database, the **manufact** table contains the names of manufacturers and their delivery times. An actual working database might contain many other attributes of a supplier, such as address and sales representative name.

The contents of **manufact** are primarily a supplement to the **stock** table. Suppose that a time-critical application frequently refers to the delivery lead time of a particular product but to no other column of **manufact**. For each such reference, the database server must read two or three pages of data to perform the lookup.

You can add a new column, **lead_time**, to the **stock** table and fill it with copies of the **lead_time** column from the corresponding rows of **manufact**. That eliminates the lookup, and so speeds up the application.

Like derived data, redundant data takes space and poses an integrity risk. In the example described in the previous paragraph, many extra copies of each manufacturer's lead time can exist. (Each manufacturer can appear in **stock** many times.) The programs that insert or update a row of **manufact** must also update multiple rows of **stock**.

The integrity risk is simply that the redundant copies of the data might not be accurate. If a lead time is changed in **manufact**, the **stock** column is outdated until it is also updated. As you do with derived data, define the conditions under which redundant data might be wrong. For more information, refer to the *Informix Guide to SQL: Syntax* and the *Informix Guide to SQL: Reference*.

Chapter

Locking

5

This chapter describes how the database server uses locks and how locks can affect performance.

This chapter discusses the following topics:

- Types of locks
- Locking during query processing
- Locking during updates, deletes, and inserts
- Monitoring and configuring locks

Lock Granularity

A *lock* is a software mechanism that prevents others from using a resource. This chapter discusses the locking mechanism placed on data. You can place a lock on the following items:

- 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, a higher probability exists that more users will 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 administrator is to increase concurrency to increase total performance without sacrificing performance for an individual user.

Row and Key Locks

Row and key locking are not the default behaviors. You must create the table with row-level locking on, as in the following example:

```
CREATE TABLE customer(customer_num serial, lname char(20)...)
    LOCK MODE ROW;
```

The ALTER TABLE statement can also change the lock mode.

When you insert or update a row, the database server creates a row lock. In some cases, you place a row lock by simply reading the row with a SELECT statement.

When you insert, update, or delete a key (performed automatically when you insert, update, or delete a row), the database server creates a lock on the key in the index.

Row and key locks generally provide the best overall performance when you are updating a relatively small number of rows because they increase concurrency. However, the database server incurs some overhead in obtaining 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 Locks

Page locking is the default behavior when you create a table without the LOCK MODE clause.

With page locking, instead of locking only the row, the database server locks the entire page that contains 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, 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 (performed automatically when you insert, update, or delete a row), the database server creates a lock on the page that contains the key in the index.

Important: *A page lock on an index page can decrease concurrency more substantially 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 user changes a large number 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 is important. For more information, refer to"Isolation Level" on page 5-8.

Table Locks

In a data warehouse environment, it might be more appropriate for queries to acquire larger granularity locks. 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 can place two types of table locks:

- Shared lock

  No users can write to the table.
- Exclusive lock

  No other users can read or write from the table.

Another 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, which informs other users 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 update most of the rows in the table, 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 multiple users.*

You can switch a table back and forth between table-level locking and the other levels of locking. This ability to switch locking levels is useful when you use a table in a data warehouse mode during certain time periods but not in others.

A transaction tells the database server to use table-level locking for a table with the LOCK TABLE statement. 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:
```

Table Locks That the Database Server Places

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 portion of the table, it places a table lock automatically instead of setting row or page locks. The database server places a table lock on a table when it creates or drops an index.

Database Locks

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
```

Waiting for Locks

When a user encounters a lock, the default behavior of a database server is to return an error to the application. You can execute the following SQL statement to wait indefinitely for a lock:

```
SET LOCK MODE TO WAIT;
```

You can also wait for a specific number of seconds, as in the following example:

```
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;
```

Locks with the SELECT Statement

The type and duration of locks that the database server places depend on the isolation level set in the application and on whether the SELECT statement is within an update cursor. The following sections explain isolation levels and update cursors.

Isolation Level

The number and duration of locks placed on data during a SELECT statement depends on the level of isolation that the user set. The type of isolation can impact overall performance because it affects concurrency.

You can set the isolation level with the SET ISOLATION or the ANSI SET TRANSACTION statement before you execute the SELECT statement. The main differences between the two statements are that SET ISOLATION has an additional isolation level, Cursor Stability, and SET TRANSACTION cannot be executed more than once in a transaction as SET ISOLATION can.

Dirty Read Isolation (ANSI Read Uncommitted)

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. With a Dirty Read, the reader can read a row that has not been committed to the database and might be eliminated 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.
```

User 2 reads row A, which user 1 rolls back seconds later. In effect, user 2 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 phantom rows are an issue only with transactions, databases that do not have logging on (and hence do not allow transactions) use Dirty Read as a default isolation level. In fact, Dirty Read is the only isolation level allowed for databases without logging on.

Committed Read Isolation (ANSI Read Committed)

Committed Read isolation removes the problem of phantom reads. A reader with this isolation level checks for locks before it returns a row. By checking for locks, the reader cannot return any uncommitted rows.

The database server does not actually place any locks for rows read during Committed Read. It simply checks for any existing rows in the internal lock table.

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 reader with Cursor Stability isolation acquires a shared lock on the row that is currently fetched. This action assures that no other user can update the row until the user fetches a new row.

The pseudocode in Figure 5-1 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 5-1
Locks Placed for Cursor Stability

```
set isolation to cursor stability
declare cursor for SELECT * from customer
open the cursor
while there are more rows
    fetch a row        ←———— Release the lock on the previous
                             row and add a lock for this row.
    do stuff
end while
close the cursor       ←———— Release the lock on the last row.
```

Repeatable Read Isolation (ANSI Serializable, ANSI Repeatable Read)

Repeatable Read isolation (ANSI Serializable and ANSI Repeatable Read) is the most strict isolation level. With Repeatable Read, the database server locks all rows examined (not just fetched) for the duration of the transaction.

The pseudocode in Figure 5-2 shows when the database server places and releases locks with a cursor.

Figure 5-2
Locks Placed for Repeatable Read

```
set isolation to repeatable read
begin work
declare cursor for SELECT * FROM customer
open the cursor
while there are more rows
    fetch a row
    do stuff
end while
close the cursor
commit work
```

fetch a row ← Add a lock for this row and every row examined to retrieve this row.

commit work ← Release all locks.

Repeatable Read is useful during any processing in which multiple rows are examined and none must change during the transaction. For example, suppose an application must check 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 scenario can occur. However, it cannot occur with Repeatable Read. The original application holds a read lock on each account that it examines until the end of the transaction, so the attempt by the second application to change the first account fails (or waits, depending upon SET LOCK MODE).

Because even examined rows are locked, if the database server reads the table sequentially, a large number of 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. If an index exists and the optimizer chooses a sequential scan instead, you can use directives to force use of the index. However, forcing a change in the query path might negatively affect query performance.

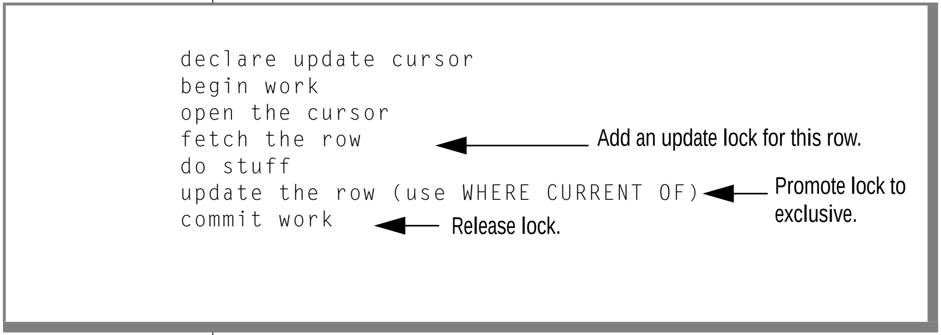
Update Cursors

An update cursor is a special kind of cursor that applications can use when the row might potentially be updated. To use an update cursor, execute SELECT FOR UPDATE in your application. Update cursors use *promotable locks*; that is, the database server places an update lock (meaning other users can still view the row) when the application fetches the row, but the lock is changed to an exclusive lock when the application updates the row using an update cursor and UPDATE...WHERE CURRENT OF.

The advantage of using an update cursor is that you can view the row with the confidence that other users cannot change it or view it with an update cursor while you are viewing it and before you update it.

In an ANSI-compliant database, update cursors are not needed because any select cursor behaves the same as an update cursor.

The pseudocode in Figure 5-3 shows when the database server places and releases locks with a cursor.



```
declare update cursor
begin work
open the cursor
fetch the row
do stuff
update the row (use WHERE CURRENT OF)
commit work
```

Figure 5-3
Locks Placed for Update Cursors

Locks Placed with INSERT, UPDATE, and DELETE

When you execute an INSERT, UPDATE, or DELETE statement, the database server uses exclusive locks. An exclusive lock means that no other users can view the 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.

When the database server removes the exclusive lock depends on the type of logging set for the database. If the database has logging, the database server removes all exclusive locks when the transaction completes (commits or rolls back). If the database does not have logging, the database server removes all exclusive locks immediately after the INSERT, UPDATE, or DELETE statement completes.

Key-Value Locking

When a user deletes a row within 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 a concept called *key-value locking* to lock the deleted row. When the database server deletes a row, key values in the indexes for the table are not removed immediately. Instead, each key value is marked as deleted, and a lock is placed on the key value.

Other users might encounter key values that are marked as deleted. The database server must determine whether a lock exists. If a lock exists, the delete has not been committed, and 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 remains unique through the end of the transaction that deleted it. Without this protection mechanism, user A might delete a unique key within a transaction and, before the transaction commits, user B might insert a row with the same key. This scenario makes 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 locks in an internal lock table. When the database server reads a row, it checks 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 must also check the lock type. The following table shows the types of locks that the lock table can contain.

Lock Type	Description	Statement Usually Placing 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 updating VARCHAR columns

In addition, the lock table might store *intent locks*, with the same lock type as previously shown. In some cases, a user might need to register its possible intent to lock an item, so that other users cannot place a lock on the item.

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 might wait for the lock to be released (if the user executed SET LOCK MODE TO WAIT). The following table shows the locks that a user can place if another user holds a certain type of lock. For example, if one user holds an exclusive lock on an item, another user requesting 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

You can view the lock table with **onstat -k**. Figure 5-4 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 5-4
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** table
- An intent-exclusive lock on the table
- An exclusive lock on the row

To determine the table to which the lock applies, execute the following SQL statement. For *tblsnum*, substitute the value shown in the **tblsnum** field in the **onstat -k** output:

```
SELECT tabname
FROM systables
WHERE partnum = hex(tblsnum)
```

For a complete definition of fields, consult your *Administrator's Guide*.

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 3-36.

To monitor the number of times that applications receive the out-of-locks error, view the **ovlock** field in the output of **onstat -p**.

If the database server commonly receives the out-of-locks error, you can increase the LOCKS parameter value. However, a very large lock table can impede performance. Although the algorithm to read the lock table is efficient, you incur some cost for reading a very large table each time that the database server reads a row. If the database server is using an unusually large number of locks, you might need to examine how individual applications are using locks.

First, monitor sessions with **onstat -u** to see if a particular user is using an especially high number of locks (a high value in the **locks** column). If a particular user 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.

In general, to reduce the number of locks placed on a table, 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 give users the impression that the application is hanging.

In Figure 5-5 on page 5-16, the **onstat -u** output shows that session ID 84 is waiting for a lock (`L` in the first column of the **Flags** field). To find out the owner of the lock, use the **onstat -k** command.

Figure 5-5
onstat -u Output That Shows Lock Usage

```
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
```

To find out the owner of the lock for which session ID 84 is waiting

1. Obtain the address of the lock in the **wait** field (300b78d8) of the **onstat -u** output.
2. Find this address (300b78d8) in the **Locks address** field of the **onstat -k** output.

   The **owner** field of this row in the **onstat -k** output contains the address of the userthread (40074140).
3. Find this address (40074140) in the **userthread** field of the **onstat -u** output.

   The **sessid** field of this row in the **onstat -u** output contains the session ID (81) that owns the lock.

To eliminate the contention problem, you can have the user exit the application gracefully. If this solution is not possible, you can kill the application process or remove the session with **onmode -z**.

Monitoring Deadlocks

A *deadlock* occurs when two users hold locks, and each user wants to acquire a lock that the other user owns.

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 potentially might wait for each other forever.

Informix 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 for any locks that the requestor holds. If so, the requestor receives an deadlock error.

Deadlock errors can be unavoidable when 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**.

In a distributed transaction, the database server does not examine lock tables from other OnLine systems, so deadlocks cannot be detected before they occur. Instead, you can set the DEADLOCK_TIMEOUT parameter. DEADLOCK_TIMEOUT specifies the number of seconds that the database server waits for a remote database server response before it returns an error. Although reasons other than a distributed deadlock might cause the delay, this mechanism keeps a transaction from hanging indefinitely.

To monitor the number of distributed deadlock timeouts, use the **dltouts** field in the **onstat -p** output.

Chapter

Fragmentation Guidelines

6

This chapter discusses the performance considerations that are involved when you use table fragmentation.

One of the most frequent causes of poor performance in relational database systems is contention for data that resides on a single I/O device. Informix database servers support table fragmentation (also *partitioning*), which allows you to store data from a single table on multiple disk devices. Proper fragmentation of high-use tables can significantly reduce I/O contention.

This chapter discusses the following topics:

- Planning a fragmentation strategy
- Designing a distribution scheme
- Distribution schemes for fragment elimination
- Fragmenting indexes
- Fragmenting temporary tables
- Improving the performance of attaching and detaching fragments
- Monitoring fragmentation

For information about fragmentation and parallel execution, refer to Chapter 9, “Parallel Database Query.”

For an introduction to fragmentation concepts and methods, refer to the *Informix Guide to Database Design and Implementation*. For information about the SQL statements that manage fragments, refer to the *Informix Guide to SQL: Syntax*.

Planning a Fragmentation Strategy

A fragmentation strategy consists of two parts:

- A distribution scheme that specifies how to group rows into fragments

  You specify the distribution scheme in the FRAGMENT BY clause of the CREATE TABLE, CREATE INDEX, or ALTER FRAGMENT statements.

- The set of dbspaces (or dbslices) in which you locate the fragments

  You specify the set of dbspaces or dbslices in the IN clause (storage option) of these SQL statements.

Formulating a fragmentation strategy requires you to make the following decisions:

1. Decide what your primary fragmentation goal is.

   Your fragmentation goals will depend, to a large extent, on the types of applications that access the table.

2. Decide how the table should be fragmented.

   You must make the following decisions:

   - Whether to fragment the table data, the table index, or both

     This decision is usually based on your primary fragmentation goal.

   - What the ideal distribution of rows or index keys is for the table

     This decision is also based on your primary fragmentation goal.

3. Decide on a distribution scheme.

   You must first choose between the following distribution schemes.

   - If you choose an expression-based distribution scheme, you must then design suitable fragment expressions.
   - If you choose a round-robin distribution scheme, the database server determines which rows to put into a specific fragment.

4. To complete the fragmentation strategy, you must decide on the number and location of the fragments:

   - The number of fragments depends on your primary fragmentation goal.
   - Where you locate fragments depends on the number of disks available in your configuration.

The following sections describe these topics:

- Fragmentation goals
- Performance-related factors for fragmentation
- Examination of your data and queries
- Physical fragmentation factors

Setting Fragmentation Goals

Analyze your application and workload to determine the balance to strike among the following fragmentation goals:

- Improved performance for individual queries

  To improve the performance of individual queries, fragment tables appropriately and set resource-related parameters to specify system resource use (memory, CPU VPs, and so forth).
- Reduced contention between queries and transactions
- If your database server is used primarily for OLTP transactions and only incidentally for decision-support queries, you can often use fragmentation to reduce contention when simultaneous queries against a table perform index scans to return a few rows.

- Increased data availability

  Careful fragmentation of dbspaces can improve data availability if devices fail. Table fragments on the failed device can be restored quickly, and other fragments are still accessible.

- Improved data-load performance

  When you use the High-Performance Loader (HPL) to load a table that is fragmented across multiple disks, it allocates threads to light append the data into the fragments in parallel. For more information on this load method, refer to the *Guide to the High-Performance Loader*.

  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 "Improving the Performance of Attaching and Detaching Fragments" on page 6-29.

The performance of a fragmented table is primarily governed by the following factors:

- The storage option that you use for allocating disk space to fragments (discussed in "Physical Fragmentation Factors" on page 6-10)
- The distribution scheme used to assign rows to individual fragments (discussed in "Designing a Distribution Scheme" on page 6-12)

Improving Performance for Individual Queries

If the primary goal of fragmentation is improved performance for individual queries, try to distribute all the rows of the table evenly over the different disks. Overall query-completion time is reduced when the database server does not have to wait for data retrieval from a table fragment that has more rows than other fragments.

If queries access data by performing sequential scans against significant portions of tables, fragment the table rows only. Do not fragment the index. If an index is fragmented and a query has to cross a fragment boundary to access the data, the performance of the query can be worse than if you do not fragment.

If queries access data by performing an index read, you can improve performance by using the same distribution scheme for the index and the table.

If you use round-robin fragmentation, do not fragment your index. Consider placing that index in a separate dbspace from other table fragments.

For more information about improving performance for queries, see "Query Expressions for Fragment Elimination" on page 6-22 and Chapter 10, "Improving Individual Query Performance."

Reducing Contention Between Queries and Transactions

Fragmentation can reduce contention for data in tables that multiple queries and OLTP applications use. Fragmentation often reduces contention when many simultaneous queries against a table perform index scans to return a few rows. For tables subjected to this type of load, fragment both the index keys and data rows with a distribution scheme that allows each query to eliminate unneeded fragments from its scan. Use an expression-based distribution scheme. For more information, refer to "Distribution Schemes for Fragment Elimination" on page 6-22.

To fragment a table for reduced contention, start by investigating which queries access which parts of the table. Next, fragment your data so that some of the queries 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 disks.

Your success in reducing contention depends on how much you know about the distribution of data in the table and the scheduling of queries against the table. For example, if the distribution of queries against the table is set up so that all rows are accessed at roughly the same rate, try to distribute rows evenly across the fragments. However, if certain values are accessed at a higher rate than others, you can compensate for this difference by distributing the rows over the fragments to balance the access rate. For more information, refer to "Designing an Expression-Based Distribution Scheme" on page 6-15.

Increasing Data Availability

When you distribute table and index fragments across different disks or devices, you improve the availability of data during disk or device failures. The database server continues to allow access to fragments stored on disks or devices that remain operational. This availability has important implications for the following types of applications:

- Applications that do not require access to unavailable fragments

  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 distribution expression uses a single column, 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. For more information, refer to "Designing an Expression-Based Distribution Scheme" on page 6-15.

- 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 require the ability to retrieve the data that is available. These applications can specify which fragments can be skipped by executing the SET DATASKIP statement before they execute a query. Alternatively, the database server administrator can specify which fragments are unavailable using the **onspaces -f** option.

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 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 are deciding how to distribute dbspaces across disks:

- **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 in a particular dbspace.
- **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. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical data.

For more information about backup and restore, see your *Backup and Restore Guide*.

Examining Your Data and Queries

To determine a fragmentation strategy, you must know how the data in a table is used. Take the following steps to gather information about a table that you might fragment.

To gather information about your table

1. Identify the queries that are critical to performance. Determine if the queries are OLTP or DSS.
2. Use the SET EXPLAIN statement to determine how the data is being accessed. For information on the output of the SET EXPLAIN statement, refer to “How to Display the Query Plan” on page 7-14. Sometimes you can determine how the data is accessed by simply reviewing the SELECT statements along with the table schema.
3. Determine what portion of the data each query examines. For example, if certain rows in the table are read most of the time, you can isolate them into a small fragment, which reduces I/O contention for other fragments.

4. Determine what statements create temporary files. Decision-support queries typically create and access large temporary files, and placement of temporary dbspaces can be critical to performance.
5. If particular tables are always joined together in a decision-support query, spread fragments for these tables across different disks.
6. Examine the columns in the table to determine which fragmentation scheme would keep each scan thread equally busy for the decision-support queries. To see how the column values are distributed, create a distribution on the column with the UPDATE STATISTICS statement and examine the distribution with **dbschema**.

```
dbschema -d database -hd table
```

Physical Fragmentation Factors

When you fragment a table, the physical placement issues pertaining to tables that are described in Chapter 4, “Table and Index Performance Considerations,” apply to individual table fragments. Because each fragment resides in its own dbspace on a disk, these issues must be addressed separately for the fragments on each disk.

Fragmented and nonfragmented tables 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 or not, Informix recommends that you create a single chunk on each disk for each dbspace.
- Extent sizes for a fragmented table are usually smaller than the extent sizes for an equivalent nonfragmented table because fragments do not grow in as large increments as the entire table. For more information on how to estimate the space to allocate, refer to “Estimating Table and Index Size” on page 4-8.

- In a fragmented table, the row ID is no longer a unique nonchanging pointer to the row on a disk. The database serve now uses the combination of fragment ID and row ID internally, inside an index, to point to the row. These two fields are unique but can change over the life of the row. An application cannot access the fragment ID; therefore, Informix recommends that you use primary keys to access a specific row in a fragmented table. For more information, refer to the *Informix Guide to Database Design and Implementation*.
- An attached index or an index on a nonfragmented table uses 4 bytes for the row ID. A detached index uses 8 bytes of disk space per key value for the fragment ID and row ID combination. For more information on how to estimate space for an index, refer to “Estimating Index Pages” on page 4-13. For more information on attached indexes and detached indexes, refer to “Fragmenting Indexes” on page 6-18

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 “Spreading Temporary Tables and Sort Files Across Multiple Disks” on page 4-7.

Designing a Distribution Scheme

After you decide whether to fragment table rows, index keys, or both, and you decide how the rows or keys should be distributed over fragments, you decide on a scheme to implement this distribution.

The database server supports the following distribution schemes:

- **Round-robin.** This type of fragmentation places rows one after another in fragments, rotating through the series of fragments to distribute the rows evenly.

  For INSERT statements, the database server uses a hash function on a random number to determine the fragment in which to place the row. For 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.
- **Expression-based.** This type of fragmentation puts rows that contain specified values in the same fragment. You specify a *fragmentation expression* that defines criteria for assigning a set of rows to each fragment, either as a range rule or some arbitrary rule. You can specify a *remainder fragment* that holds all rows that do not match the criteria for any other fragment although a remainder fragment reduces the efficiency of the expression-based distribution scheme.

Choosing a Distribution Scheme

Figure 6-1 compares round-robin and expression-based distribution schemes for three important features.

Figure 6-1
Distribution-Scheme Comparisons

Distribution Scheme	Ease of Data Balancing	Fragment Elimination	Data Skip
Round-robin	Automatic. Data is balanced over time.	The database server cannot eliminate fragments.	You cannot determine if the integrity of the transaction is compromised when you use the data-skip feature. However, you can insert into a table fragmented by round-robin.
Expression-based	Requires knowledge of the data distribution.	If expressions on one or two column are used, the database server can eliminate fragments for queries that have either range or equality expressions.	You can determine whether the integrity of a transaction has been compromised when you use the data-skip feature. You cannot insert rows if the appropriate fragment for those rows is down.

The distribution scheme that you choose depends on the following factors:

- The features that you want to take advantage of in Figure 6-1
- Whether or not your queries tend to scan the entire table
- Whether or not you know the distribution of data to be added
- Whether or not your applications tend to delete many rows
- Whether or not you cycle your data through the table

Basically, the round-robin scheme provides the easiest and surest way of balancing data. However, with round-robin distribution, you have no information about the fragment in which a row is located, and the database server cannot eliminate fragments.

In general, round-robin is the correct choice 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. (If they do, load balancing could be degraded.)

An expression-based scheme might be the best choice to fragment the data if any of the following conditions apply:

- Your application calls for numerous decision-support queries that scan specific portions of the table.
- You know what the data distribution is.
- You plan to cycle data through a database.

If you plan to add and delete large amounts of data periodically based on the value of a column such as date, you can 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 other users to access. Only the fragment that you attach or detach is not available to other users.
- With the performance enhancements, 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 "Improving the Performance of Attaching and Detaching Fragments" on page 6-29.

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 6-18.

Designing an Expression-Based Distribution Scheme

The first step in designing 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 this information, run the UPDATE STATISTICS statement for the table and then use the **dbschema** utility to examine the distribution.

Once you know the data distribution, you can design a fragmentation rule that distributes data across fragments as required to meet your fragmentation goal. If your primary goal is to improve performance, your fragment expression should generate an even distribution of rows across fragments.

If your primary fragmentation goal is improved concurrency, analyze the queries that access the table. If certain rows are accessed at a higher rate than others, you can compensate for this by opting for an uneven distribution of data over the fragments that you create.

Try not to use columns that are subject to frequent updates in the distribution expression. Such updates can cause rows to move from one fragment to another (that is, be deleted from one and added to another), and this activity increases CPU and I/O overhead.

Try to create nonoverlapping regions based on a single column with no REMAINDER fragment for the best fragment-elimination characteristics. Dynamic Server eliminates fragments from query plans whenever the query optimizer can determine that the values selected by the WHERE clause do not reside on those fragments, based on the expression-based fragmentation rule by which you assign rows to fragments. For more information, refer to "Distribution Schemes for Fragment Elimination" on page 6-22.

Suggestions for Improving Fragmentation

The following suggestions are guidelines for fragmenting tables and indexes:

- For optimal performance in decision-support queries, fragment the table to increase parallelism, but do not fragment the indexes. Detach the indexes, and place them in a separate dbspace.
- For best performance in OLTP, use fragmented indexes to reduce contention between sessions. You can often fragment an index by its key value, which means the OLTP query only has to look at one fragment to find the location of the row.

  If the key value does not reduce contention, as when every user looks at the same set of key values (for instance, a date range), consider fragmenting the index on another value used in the WHERE clause. To cut down on fragment administration, consider not fragmenting some indexes, especially if you cannot find a good fragmentation expression to reduce contention.
- Use round-robin fragmentation on data when the table is read sequentially by decision-support queries. Round-robin fragmentation is a good method for spreading data evenly across disks when no column in the table can be used for an expression-based fragmentation scheme. However, in most DSS queries, all fragments are read.
- If you are using expressions, create them so that I/O requests, rather than quantities of data, are balanced across disks. For example, if the majority of your queries access only a portion of data in the table, set up your fragmentation expression to spread active portions of the table across disks even if this arrangement results in an uneven distribution of rows.
- Keep fragmentation expressions simple. Fragmentation expressions can be as complex as you want. However, complex expressions take more time to evaluate and might prevent fragments from being eliminated from queries.

- Arrange fragmentation expressions so that the most-restrictive condition for each dbspace is tested within the expression first. When Dynamic Server tests a value against the criteria for a given fragment, evaluation stops when a condition for that fragment tests false. Thus, if the condition that is most likely to be false is placed first, fewer conditions need to be evaluated before Dynamic Server moves to the next fragment. For example, in the following expression, Dynamic Server tests all six of the inequality conditions when it attempts to insert a row with a value of `25`:

  ```
  x >= 1 and x <= 10 in dbspace1,
  x > 10 and x <= 20 in dbspace2,
  x > 20 and x <= 30 in dbspace3
  ```

  By comparison, only four conditions in the following expression need to be tested: the first inequality for dbspace1 (`x <= 10`), the first for dbspace2 (`x <= 20`), and both conditions for dbspace3:

  ```
  x <= 10 and x >= 1 in dbspace1,
  x <= 20 and x > 10 in dbspace2,
  x <= 30 and x > 20 in dbspace3
  ```

- Avoid any expression that requires a data-type conversion. Type conversions increase the time to evaluate the expression. For instance, a DATE data type is implicitly converted to INTEGER for comparison purposes.
- Do not fragment on columns that change frequently unless you are willing to incur the administration costs. For example, if you fragment on a date column and older rows are deleted, the fragment with the oldest dates tends to empty, and the fragment with the recent dates tends to fill up. Eventually you have to drop the old fragment and add a new fragment for newer orders.
- Do not fragment every table. Identify the critical tables that are accessed most frequently. Put only one fragment for a table on a disk.
- Do not fragment small tables. Fragmenting a small table across many disks might not be worth the overhead of starting all the scan threads to access the fragments. Also, balance the number of fragments with the number of processors on your system.
- When you define a fragmentation strategy on an unfragmented table, check your next-extent size to ensure that you are not allocating large amounts of disk space for each fragment.

Fragmenting Indexes

When you fragment a table, the indexes that are associated with that table are fragmented implicitly, according to the fragmentation scheme that you use. You can also use the FRAGMENT BY EXPRESSION clause of the CREATE INDEX statement to fragment the index for any table explicitly. Each index of a fragmented table occupies its own tblspace with its own extents.

You can fragment the index with either of the following strategies:

- Same fragmentation strategy as the table
- Different fragmentation strategy from the table

Attached Indexes

An *attached index* is an index that implicitly follows the table fragmentation strategy (distribution scheme and set of dbspaces in which the fragments are located).

The database server creates an attached index automatically when you first fragment the table.

To create an attached index, do not specify a fragmentation strategy, as in the following sample SQL statements:

```
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);
```

The database server fragments the attached index according to the same distribution scheme as the table by using 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 tblspace as the corresponding table data. Therefore, the data and index pages can be interleaved within the tblspace.
- An attached index or an index on a nonfragmented table uses 4 bytes for the row ID for each index entry. For more information on how to estimate space for an index, refer to "Estimating Index Pages" on page 4-13.

Detached Indexes

A *detached index* is an index with a separate fragmentation strategy that you set up explicitly with the CREATE INDEX statement, as in the following sample SQL statements:

```
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;
```

This example illustrates a common fragmentation strategy to fragment indexes in the same way as the tables but to specify different dbspaces for the index fragments. This fragmentation strategy of putting the index fragments in different dbspaces from the table can improve the performance of operations such as backup, recovery, and so forth.

If you do not want to fragment the index, you can put the entire index in a separate dbspace.

You can fragment the index for any table by expression. However, you cannot explicitly create a round-robin fragmentation scheme for an index. Whenever you fragment a table using a round-robin fragmentation scheme, Informix recommends that you convert all indexes that accompany the table to detached indexes for the best performance.

Detached indexes have the following physical characteristics:

- Each detached index fragment resides in a different tblspace from the corresponding table data. Therefore, the data and index pages *cannot* be interleaved within the tblspace.
- Attached index fragments have their own extents and *tblspace IDs*. The tblspace ID is also known as the *fragment ID* and *partition number*. A detached index uses 8 bytes of disk space per index entry for the fragment ID and row ID combination. For more information on how to estimate space for an index, refer to "Estimating Index Pages" on page 4-13.

The database server stores the location of each table and index fragment, along with other related information, in the system catalog table **sysfragments**. You can use the **sysfragments** system catalog table to access information such as the following about fragmented tables and indexes:

- The value in the **partn** field is the partition number or fragment ID of the table or index fragment. The partition number for a detached index is different from the partition number of the corresponding table fragment.
- The value in the **strategy** field is the distribution scheme used in the fragmentation strategy.

For a complete description of field values that this **sysfragments** system catalog table contains, refer to the *Informix Guide to SQL: Reference*. For information on how to use **sysfragments** to monitor your fragments, refer to "Monitoring Fragment Use" on page 6-40.

Restrictions on Indexes for Fragmented Tables

If the database server scans a fragmented index, multiple index fragments must be scanned and the results merged together. (The exception is if the index is fragmented according to some index-key range rule, and the scan does not cross a fragment boundary.) Because of this requirement, performance on index scans might suffer if the index is fragmented.

Because of these performance considerations, the database server places the following restrictions on indexes:

- You cannot fragment indexes by round-robin.
- You cannot fragment unique indexes by an expression that contains columns that are not in the index key.

For example, the following statement is not valid:

```
CREATE UNIQUE INDEX ia on tab1(col1)
    FRAGMENT BY EXPRESSION
        col2<10 in dbsp1,
        col2>=10 AND col2<100 in dbsp2,
        col2>100 in dbsp3;
```

Fragmenting a Temporary Table

You can fragment an explicit temporary table across dbspaces reside on different disks. For more information on explicit and implicit temporary tables, refer to your *Administrator's Guide*.

You can create a temporary, fragmented table with the TEMP TABLE clause of the CREATE TABLE statement. However, you cannot alter the fragmentation strategy of fragmented temporary tables (as you can with permanent tables).

The database server deletes the fragments that are created for a temporary table at the same time that it deletes the temporary table.

You can define your own fragmentation strategy for an explicit temporary table, or you can let the database server dynamically determine the fragmentation strategy.

Distribution Schemes for Fragment Elimination

Fragment elimination is a database server feature that reduces the number of fragments involved in a database operation. This capability can improve performance significantly and reduce contention for the disks on which fragments reside.

Fragment elimination improves both response time for a given query and concurrency between queries. Because the database server does not need to read in unnecessary fragments, I/O for a query is reduced. Activity in the LRU queues is also reduced.

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.

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

Query Expressions for Fragment Elimination

A query expression (the expression in the WHERE clause) can consist of any of the following expressions:

- Simple expression
- Not simple expression
- Multiple expressions

The database server considers only simple expressions or multiple simple expressions combined with certain operators for fragment elimination.

A simple expression consists of the following parts:

```
column operator value
```

Simple Expression Part	Description
`column`	Is a single column name. Dynamic Server supports fragment elimination on all column types except columns that are defined with the NCHAR, NVARCHAR, BYTE, and TEXT data types.
`operator`	Must be an equality or range operator.
`value`	Must be a literal or a host variable.

The following examples show simple expressions:

```
name = "Fred"
date < "01/25/1994"
value >= :my_val
```

The following examples are not simple expressions:

```
unitcost * count > 4500
price <= avg(price)
result + 3 > :limit
```

The database server considers two types of simple expressions for fragment elimination, based on the operator:

- Range expressions
- Equality expressions

Range Expressions in Query

Range expressions use the following relational operators:

```
<, >, <=, >=, !=
```

Dynamic Server can handle one or two column fragment elimination on queries with any combination of these relational operators in the WHERE clause.

Dynamic Server can also eliminate fragments when these range expressions are combined with the following operators:

```
AND, OR, NOT
IS NULL, IS NOT NULL
MATCH, LIKE
```

If the range expression contains MATCH or LIKE, Dynamic Server can also eliminate fragments if the string ends with a wildcard character. The following examples show query expressions that can take advantage of fragment elimination:

```
columna MATCH "ab*"
columna LIKE "ab%" OR columnb LIKE "ab*"
```

Equality Expressions in Query

Equality expressions use the following equality operators:

```
=, IN
```

Dynamic Server can handle one or multiple column fragment elimination on queries with a combination of these equality operators in the WHERE clause. Dynamic Server can also eliminate fragments when these equality expressions are combined with the following operators:

```
AND, OR
```

Effectiveness of Fragment Elimination

Dynamic Server cannot eliminate fragments when you fragment a table with a round-robin distribution scheme. Furthermore, not all expression-based distribution schemes give you the same fragment-elimination behavior.

Figure 6-2 summarizes the fragment-elimination behavior for different combinations of expression-based distribution schemes and query expressions.

Figure 6-2
Fragment Elimination for Different Categories of Expression-Based Distribution Schemes and Query Expressions

Type of Query (WHERE clause) Expression	**Type of Expression-Based Distribution Scheme**		
	Nonoverlapping Fragments on a Single Column	**Overlapping or Non-contiguous Fragments on a Single Column**	**Nonoverlapping Fragments on Multiple Columns**
Range expression	Fragments can be eliminated.	Fragments cannot be eliminated.	Fragments cannot be eliminated.
Equality expression	Fragments can be eliminated.	Fragments can be eliminated.	Fragments can be eliminated.

Figure 6-2 indicates that the distribution schemes enable fragment elimination, but the effectiveness of fragment elimination is determined by the WHERE clause of the query in question.

For example, consider a table fragmented with the following expression:

```
FRAGMENT BY EXPRESSION
100 < column_a AND column_b < 0 IN dbsp1,
100 >= column_a AND column_b < 0 IN dbsp2,
column_b >= 0 IN dbsp3
```

Dynamic Server cannot eliminate any fragments from the search if the WHERE clause has the following expression:

```
column_a = 5 OR column_b = -50
```

On the other hand, Dynamic Server can eliminate the fragment in dbspace **dbsp3** if the WHERE clause has the following expression:

```
column_b = -50
```

Furthermore, Dynamic Server can eliminate the two fragments in dbspaces **dbsp2** and **dbsp3** if the WHERE clause has the following expression:

```
column_a = 5 AND column_b = -50
```

The following sections discuss distribution schemes to fragment data to improve fragment elimination behavior.

Nonoverlapping Fragments on a Single Column

A fragmentation rule that creates nonoverlapping fragments on a single column is the preferred fragmentation rule from a fragment-elimination standpoint. The advantage of this type of distribution scheme is that Dynamic Server can eliminate fragments for queries with range expressions as well as queries with equality expressions. Informix recommends that you meet these conditions when you design your fragmentation rule. Figure 6-3 gives an example of this type of fragmentation rule.

```
.
.
.
FRAGMENT BY EXPRESSION
a <= 8 OR a IN (9,10) IN dbsp1,
10 < a <= 20 IN dbsp2,
a IN (21,22, 23) IN dbsp3,
a > 23 IN dbsp4;
```

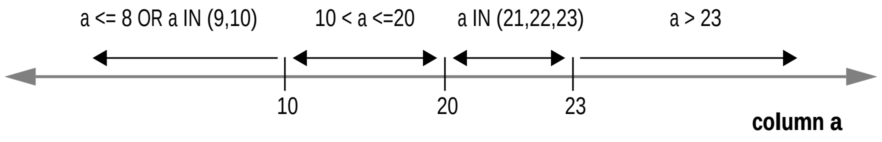


Figure 6-3
Schematic Example of Nonoverlapping Fragments on a Single Column

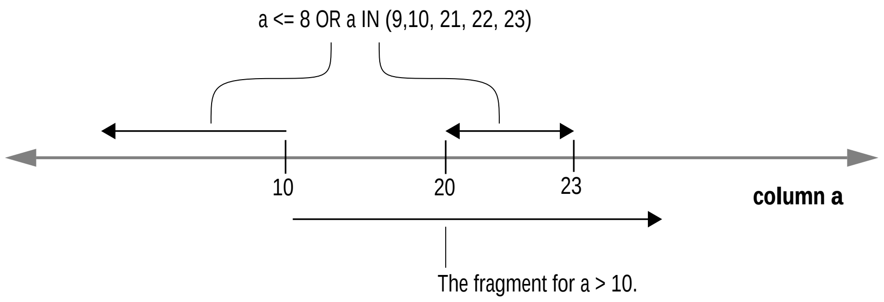
You can create nonoverlapping fragments using a range rule or an arbitrary rule based on a single column. You can use relational operators, as well as AND, IN, OR, or BETWEEN. Be careful when you use the BETWEEN operator. When Dynamic Server parses the BETWEEN keyword, it includes the endpoints that you specify in the range of values. Avoid using a REMAINDER clause in your expression. If you use a REMAINDER clause, Dynamic Server cannot always eliminate the remainder fragment.

Overlapping Fragments on a Single Column

The only restriction for this category of fragmentation rule is that you base the fragmentation rule on a single column. The fragments can be overlapping and noncontiguous. You can use any range, MOD function, or arbitrary rule that is based on a single column. Figure 6-4 gives an example of this type of fragmentation rule.

Figure 6-4
Schematic Example of Overlapping Fragments on a Single Column

```
.
.
.
FRAGMENT BY EXPRESSION
a <= 8 OR a IN (9,10,21,22,23) IN dbsp1,
a > 10 IN dbsp2;
```



If you use this type of distribution scheme, Dynamic Server can eliminate fragments on an equality search but not a range search. This distribution scheme can still be very useful because all INSERT and many UPDATE operations perform equality searches.

This alternative is acceptable if you cannot use an expression that creates nonoverlapping fragments with contiguous values. For example, in cases where a table is growing over time, you might want to use a MOD function rule to keep the fragments of similar size. Expression-based distribution schemes that use MOD function rules fall into this category because the values in each fragment are not contiguous.

Nonoverlapping Fragments, Multiple Columns

This category of expression-based distribution scheme uses an arbitrary rule to define nonoverlapping fragments based on multiple columns. Figure 6-5 gives an example of this type of fragmentation rule.

```
.
.
.

FRAGMENT BY EXPRESSION
0 < a <= 10 AND b IN ('E', 'F','G') IN dbsp1,
0 < a <= 10 AND b IN ('H', 'I','J') IN dbsp2,
10 < a <= 20 AND b IN ('E', 'F','G') IN dbsp3,
10 < a <= 20 AND b IN ('H', 'I','J') IN dbsp4,
20 < a <= 30 AND b IN ('E', 'F','G') IN dbsp5,
20 < a <= 30 AND b IN ('H', 'I','J') IN dbsp6;
```

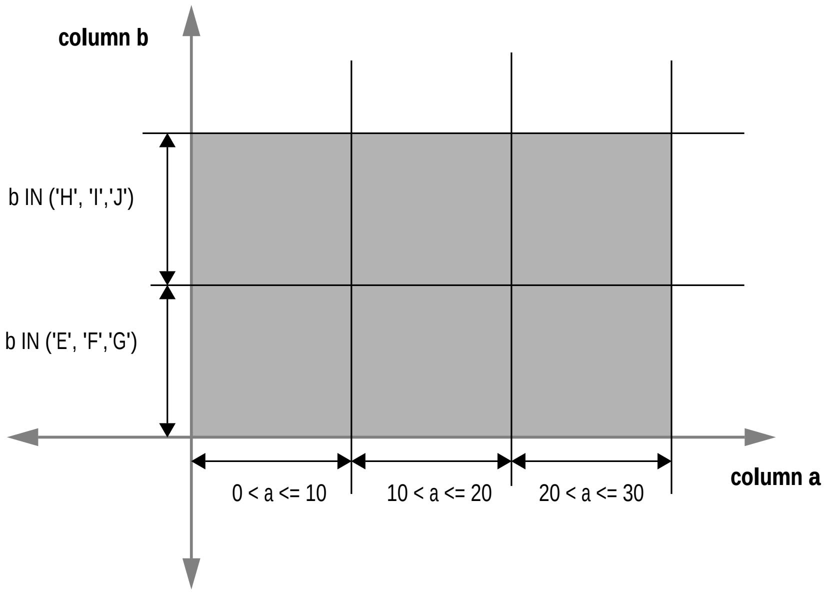


Figure 6-5
Schematic Example of Nonoverlapping Fragments on Two Columns

If you use this type of distribution scheme, Dynamic Server can eliminate fragments on an equality search but not a range search. Again, this capability can still be very useful because all INSERT operations and many UPDATE operations perform equality searches. Avoid using a REMAINDER clause in the expression. If you use a REMAINDER clause, Dynamic Server cannot always eliminate the remainder fragment.

This alternative is acceptable if you cannot obtain sufficient granularity using an expression based on a single column.

Improving the Performance of Attaching and Detaching Fragments

Many users 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 technology. However, the ALTER FRAGMENT ATTACH and DETACH statements can take a long time to execute when the database server rebuilds indexes on the surviving table.

Dynamic Server provides performance optimizations for the ALTER FRAGMENT ATTACH and DETACH statements that cause the database server to reuse the indexes on the surviving tables. Therefore, the database server can eliminate the index build during the attach or detach operation, which:

- 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 take advantage of these performance optimizations for the ALTER FRAGMENT ATTACH statement, you must meet all of the following requirements:

- Formulate appropriate distribution schemes for your table and index fragments.
- Ensure that there is no data movement between the resultant partitions due to fragment expressions.
- Update statistics for all the participating tables.
- Make the indexes on the attached tables unique if the index on the surviving table is unique.

Important: *Only logging databases can benefit from the performance improvements for the ALTER FRAGMENT ATTACH statement. Without logging, the database server works with multiple copies of the same table to ensure recoverability of the data when a failure happens. This requirement prevents reuse of the existing index fragments.*

Formulating Appropriate Distribution Schemes

This section describes three distribution schemes that allow the attach operation of the ALTER FRAGMENT statement to reuse existing indexes:

- Fragment the index in the same way as the table.
- Fragment the index with the identical set of fragment expressions as the table.
- Attach unfragmented tables to form a fragmented table.

Fragmenting the Index in the Same Way as the Table

You fragment an index in the same way as the table when you create an index without specifying a fragmentation strategy. As "Planning a Fragmentation Strategy" on page 6-4 describes, a fragmentation strategy is the distribution scheme and set of dbspaces in which the fragments are located.

For example, suppose 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;

CREATE INDEX idx1 ON tb1(a);
```

Now, suppose you create another table that is not fragmented and you subsequently decide to attach it to the fragmented table:

```
CREATE TABLE tb2 (a int, CHECK (a >=10 AND a<15))
    IN db3;

CREATE INDEX idx2 ON tb2(a)
    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 **idx2** if no data movement occurs between the existing and the new table fragments. If no data movement occurs:

- The database server reuses index **idx2** and converts it to a fragment of index **idx1**.
- The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**.

If the database server discovers that one or more rows in the table **tb2** belong to preexisting fragments of the table **tb1**, the database server:

- drops and rebuilds the index **idx1** to include the rows that were originally in tables **tb1** and **tb2**.
- drops the index **idx2**.

For more information on how to ensure no data movement between the existing and the new table fragments, refer to "Ensuring No Data Movement When You Attach a Fragment" on page 6-34.

Fragmenting the Index with the Identical Distribution Scheme as the Table

You fragment an index with the same distribution scheme as the table when you create the index using the identical fragment expressions as the table.

The database server determines if the fragment expressions are identical based on the equivalency of the expression tree instead of the algebraic equivalence. For example, consider the following two expressions:

```
        (col1 >= 5)
        (col1 = 5 OR col1 > 5)
```

Although these two expressions are algebraically equivalent, they are not identical expressions.

Suppose you create two fragmented tables and indexes with the following SQL statements:

```
    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;

    CREATE TABLE tb2 (a INT CHECK a> 30 AND a<= 40)
        IN tabdbspc4;
    CREATE INDEX idx2 ON tb2(a)
        IN idxdbspc4;
```

Now, suppose you attach table **tb2** to table **tb1** with the following sample SQL statement:

```
    ALTER FRAGMENT ON TABLE tb1
            ATTACH tb2 AS (a <= 40);
```

The database server can eliminate the rebuild of index **idx1** for this attach operation for the following reasons:

- The fragmentation expression for index **idx1** is identical to the fragmentation expression for table **tb1**. The database server:
  - expands the fragmentation of the index **idx1** to the dbspace **idxdbspc4**
  - converts index **idx2** to a fragment of index **idx1**.
- No rows move from one fragment to another because the CHECK constraint is identical to the resulting fragmentation expression of the attached table.

  For more information on how to ensure no data movement between the existing and the new table fragments, refer to "Ensuring No Data Movement When You Attach a Fragment" on page 6-34.

Attaching Unfragmented Tables Together

You also take advantage of the performance improvements for the ALTER FRAGMENT ATTACH operation when you combine two unfragmented tables into one fragmented table.

For example, suppose you create two unfragmented tables and indexes with the following SQL statements:

```
create table tb1(a int) in db1;
    create index idx1 on tb1(a) in db1;
create table tb2(a int) in db2;
    create index idx2 on tb2(a) in db2;
```

You might want to combine these two unfragmented tables with the following sample distribution scheme:

```
ALTER FRAGMENT ON TABLE tb1
    ATTACH
        tb1 AS (a <= 100),
        tb2 AS (a > 100);
```

If there is no data migration between the fragments of **tb1** and **tb2**, the database server redefines index **idx1** with the following fragmentation strategy:

```
CREATE INDEX idx1 ON tb1(a) F
    FRAGMENT BY EXPRESSION
        a <= 100 IN db1,
        a > 100 IN db2;
```

Important: *This behavior results in a different fragmentation strategy for the index than in earlier versions of the database server. In previous versions of the database server, the ALTER FRAGMENT ATTACH statement creates an unfragmented detached index in the dbspace **db1**.*

Ensuring No Data Movement When You Attach a Fragment

You can ensure that no data movement occurs by taking the following steps:

1. Establish a check constraint on the attached table that is identical to the fragment expression that it will assume after the ALTER FRAGMENT ATTACH operation.
2. Define the fragments with nonoverlapping expressions.

For example, you might 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;

CREATE INDEX idx1 ON tb1(a);
```

Suppose you create another table that is not fragmented, and you subsequently decide to attach it to the fragmented table.

```
CREATE TABLE tb2 (a int, check (a >=10 and a<15))
    IN db3;

CREATE INDEX idx2 ON tb2(a)
    IN db3;

ALTER FRAGMENT ON TABLE tb1
        ATTACH
            tb2 AS (a >= 10 and a<15) AFTER db2;
```

This ALTER FRAGMENT ATTACH operation takes advantage of the existing index **idx2** because the following steps were performed in the example to prevent data movement between the existing and the new table fragment:

- The check constraint expression in the CREATE TABLE **tb2** statement is identical to the fragment expression for table **tb2** in the ALTER FRAGMENT ATTACH statement.
- The fragment expressions specified in the CREATE TABLE **tb1** and the ALTER FRAGMENT ATTACH statements are not overlapping.

Therefore, the database server preserves index **idx2** in dbspace **db3** and converts it into a fragment of index **idx1**. The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**.

Updating Statistics on All Participating Tables

The database server tries to reuse the indexes on the attached tables as fragments of the resultant index. However, the corresponding index on the attached table might not exist, or it is not usable due to disk-format mismatches. In these cases, it might be faster to build an index on the attached tables rather than to build the entire index on the resultant table.

The database server estimates the cost to create the whole index on the resultant table. The database server then compares this cost to the cost of building the individual index fragments for the attached tables and chooses the index build with the least cost.

To ensure the correctness of the cost estimates, Informix recommends that you execute the UPDATE STATISTICS statement on all of the participating tables before you attach the tables. The LOW mode of the UPDATE STATISTICS statement is sufficient to derive the appropriate statistics for the optimizer to determine costs estimates for rebuilding indexes.

Corresponding Index Does Not Exist

Suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int, b 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 you create two more tables that are not fragmented, and you subsequently decide to attach them to the fragmented table.

```
CREATE TABLE tb2 (a int, b int, check (a >=10 and a<15))
    IN db3;
CREATE INDEX idx2 ON tb2(a)
    IN db3;

CREATE TABLE tb3 (a int, b int, check (a >= 15 and a<20))
    IN db4;
CREATE INDEX idx3 ON tb3(b)
    IN db4;

UPDATE STATISTICS FOR TABLE tb1;
UPDATE STATISTICS FOR TABLE tb2;
UPDATE STATISTICS FOR TABLE tb3;

ALTER FRAGMENT ON TABLE tb1
    ATTACH
        tb2 AS (a >= 10 and a<15)
        tb3 AS (a >= 15 and a<20);
```

In the preceding example, table **tb3** does not have an index on column **a** that can serve as the fragment of the resultant index **idx1**. The database server estimates the cost of building the index fragment for column **a** on the consumed table **tb3** and compares this cost to rebuilding the entire index for all fragments on the resultant table. The database server chooses the index build with the least cost.

Index on Table Is Not Usable

Suppose you create tables and indexes as in the previous section, but the index on the third table specifies a dbspace that is also used by the first table. The following SQL statements show this scenario:

```
CREATE TABLE tb1(a int, b int)
    FRAGMENT BY EXPRESSION
        (a >=0 and a < 5) IN db1,
        (a >=5 and a <10) IN db2;
CREATE INDEX idx1 ON tb1(a);
CREATE TABLE tb2 (a int, b int, check (a >=10 and a<15))
    IN db3;
CREATE INDEX idx2 ON tb2(a)
    IN db3;

CREATE TABLE tb3 (a int, b int, check (a >= 15 and a<20))
    IN db4;
CREATE INDEX idx3 ON tb3(a)
    IN db2 ;
```

This example creates the index **idx3** on table **tb3** in the dbspace **db2**. As a result, index **idx3** is not usable because index **idx1** already has a fragment in the dbspace **db2,** and the fragmentation strategy does not allow more than one fragment to be specified in a given dbspace.

Again, the database server estimates the cost of building the index fragment for column **a** on the consumed table **tb3** and compares this cost to rebuilding the entire index **idx1** for all fragments on the resultant table. Then, the database server chooses the index build with the least cost.

Improving ALTER FRAGMENT DETACH Performance

You can take advantage of the performance improvements for the ALTER FRAGMENT DETACH statement by formulating 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 use one of the following fragmentation strategies:

- Fragment the index the same as the table.
- Fragment the index with the identical distribution scheme as the table.

***Important:** Only logging databases can benefit from the performance improvements for the ALTER FRAGMENT DETACH statement. Without logging, the database server works with multiple copies of the same table to ensure recoverability of the data when a failure happens. This requirement prevents reuse of the existing index fragments.*

Fragmenting the Index in the Same Way as the Table

You fragment an index in the same way as the table when you create a fragmented table and subsequently create an index without specifying a fragmentation strategy.

For example, suppose 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 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 the table, the ALTER FRAGMENT DETACH statement does not rebuild the index after the detach operation. The database server drops the fragment of the index in dbspace **db3**, updates the system catalogs, and eliminates the index build.

Fragmenting the Index with the Same Distribution Scheme as the Table

You fragment an index with the same distribution scheme as the table when you create the index using the same fragment expressions as the table.

A common fragmentation strategy is to fragment indexes in the same way as the tables but to specify different dbspaces for the index fragments. This fragmentation strategy of putting the index fragments into different dbspaces from the table can improve the performance of operations such as backup, recovery, and so forth.

For example, suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int, b 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)
    FRAGMENT BY EXPRESSION
        (a >=0 and a< 5) IN db4,
        (a >=5 and a< 10) IN db5,
        (a >=10 and a<15) IN db6;
```

Now, suppose 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 distribution scheme of the index is the same as the table, the ALTER FRAGMENT DETACH statement does not rebuild of the index after the detach operation. The database server drops the fragment of the index in dbspace **db3**, updates the system catalogs, and eliminates the index build.

Monitoring Fragment Use

Once you determine a fragmentation strategy, you can monitor fragmentation in the following ways:

- Run individual **onstat** utility commands to capture information about specific aspects of a running query.
- Execute a SET EXPLAIN statement before you run a query to write the query plan to an output file.

Using the onstat Utility

You can monitor I/O activity to verify your strategy and determine whether I/O is balanced across fragments.

The **onstat -g ppf** command displays the number of read-and-write requests sent to each fragment that is currently open. Although these requests do not indicate how many individual disk I/O operations occur (a request can trigger multiple I/O operations), you can get a good idea of the I/O activity from these columns.

However, the output by itself does not show in which table a fragment is located. You can determine the table for the fragment by joining the **partnum** column in the output to the **partnum** column in the **sysfragments** system catalog table. The **sysfragments** table displays the associated **table id**. You can determine the table name for the fragment by joining the **table id** column in **sysfragments** to the **table id** column in **systables**.

To determine the table name

1. Obtain the value in the **partnum** field of the **onstat -g ppf** 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 **systables**. 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;
```

Using SET EXPLAIN

When the table is fragmented, the output of the SET EXPLAIN ON statement shows which table or index the database server scans to execute the query. The SET EXPLAIN output identifies the fragments with a fragment number. The fragment numbers are the same as those contained in the **partn** column in the **sysfragments** system catalog table.

The following example of SET EXPLAIN output shows a query that takes advantage of fragment elimination and scans two fragments in table **t1**:

```
QUERY:
------
select * from t1 where c1 > 12

Estimated Cost: 3
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Serial, fragments: 1, 2)

    Filters: informix.t1.c1 > 12
```

If the optimizer must scan all fragments (that is, if it is unable to eliminate any fragment from consideration), the SET EXPLAIN output displays `fragments: ALL`. In addition, if the optimizer eliminates all the fragments from consideration (that is, none of the fragments contain the queried information), the SET EXPLAIN output displays `fragments: NONE`. For information on how the database server eliminates a fragment from consideration, refer to "Distribution Schemes for Fragment Elimination" on page 6-22.

For more information on the SET EXPLAIN ON statement, refer to "How to Display the Query Plan" on page 7-14.

Chapter

7

Queries and the Query Optimizer

This chapter explains how the database server manages query optimization. It discusses the following topics:

- The query plan
- Factors that affect the query plan
- Operations that take the most time when the database server processes a query
- Optimization of stored procedures

The parallel database query (PDQ) features in the database server provide the largest potential performance improvements for a query. PDQ provides the most substantial performance gains if you fragment your tables as described in Chapter 6, "Fragmentation Guidelines."

Chapter 9, "Parallel Database Query," describes the Memory Grant Manager (MGM) and explains how to control resource use by queries. Chapter 10, "Improving Individual Query Performance," explains how to improve the performance of specific queries.

The 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, sort-merge, or nested loop), and the order in which tables are evaluated or joined. The components of a query plan are explained in detail in the following section.

Access Plan

The way that the optimizer chooses to read a table is called an *access plan*. The simplest method to access a table is to read it sequentially, called a *table scan*. The optimizer chooses a table scan when most of the table must be read, or 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 column in the index is the same as a column in a filter of the query, the optimizer can use the index to retrieve only the rows required by the query. The optimizer can use a *key-only index scan* if the columns requested are within one index on the table. The database server retrieves the needed data from the index and does not access the associated table.

The optimizer compares the cost of each plan to determine the best one. The database server derives cost from estimates of the number of I/O operations required, calculations to produce the results, rows accessed, sorting, and so forth.

Join Plan

When a query contains more than one table, they are usually joined together by filters 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";
```

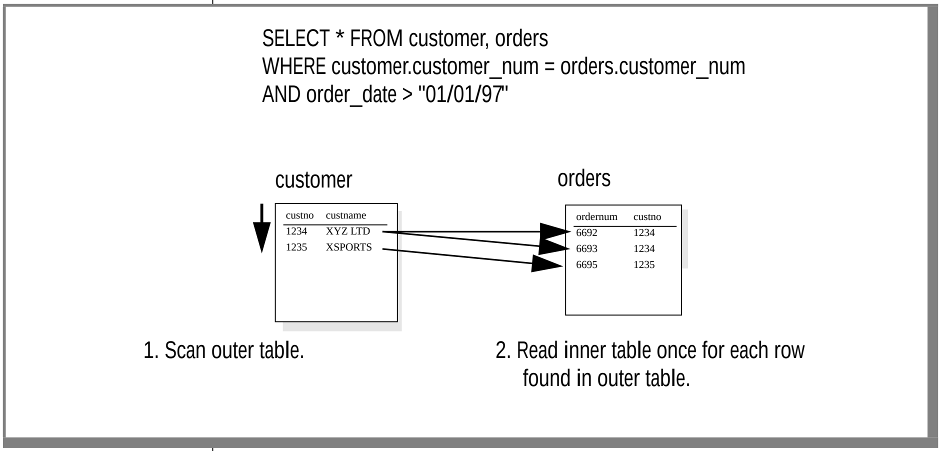
The way that the optimizer chooses to join the tables is the *join plan*. The join method can be a nested-loop join or a hash join.

Because of the nature of hash joins, an application with isolation level 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 leads to decreased concurrency among connections. Conversely, nested-loop joins lock fewer records but provide reduced 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*, then joins each of the rows that pass table filters to the rows found in the second, or *inner table* (refer to Figure 7-1). The outer table can be accessed by an index or by a table scan. The database server applies any table filters first. For each row that satisfies the filters on the outer 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.



```
SELECT * FROM customer, orders
WHERE customer.customer_num = orders.customer_num
AND order_date > "01/01/97"
```

Figure 7-1
Nested-Loop Join

If the inner table does not have an index, the database server might construct an *autoindex* at the time of query execution. The optimizer might determine that the cost to construct an *autoindex* at the time of query execution is less than the cost to scan the inner table for each qualifying row in the outer table.

If the optimizer changes a subquery to a nested-loop join, it might use a variation of the nested-loop join, called a *semi join*. In a semi join, the database server reads the inner table only until it finds a match. In other words, for each row in the outer table, the inner table contributes at most one row. For more information on how the optimizer handles subqueries, refer to "Query Plans for Subqueries" on page 7-9.

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 both tables. No index and no sorting is required when the database server performs a hash join.

A hash join consists of two activities: building the hash table (*build* phase) and probing the hash table (*probe* phase). Figure 7-2 shows the hash join in more detail.

In the build phase, the database server reads one table and, after it applies any filters, creates a hash table. You can think of a hash table conceptually as a series of *buckets*, each with an address that is derived from the key value by applying a hash function. The database server does not sort keys in a particular hash bucket.

Smaller hash tables can fit in the virtual portion of Dynamic Server shared memory. The database server stores larger hash files on disk in the dbspace specified by the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

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 hash function on the key and probes the hash table to find a match.

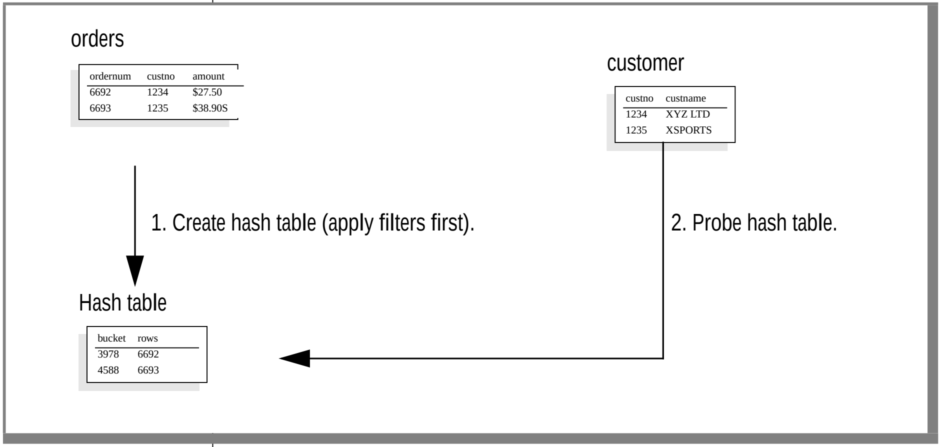


Figure 7-2
How a Hash Join Is Executed

Join Order

The order that tables are joined in a query is extremely important. A poor join order can cause query performance to decline noticeably.

The following SELECT statement 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**.

For an example of how the database server executes a plan according to a specific join order, refer to "An Example of How Query Plans Are Executed" on page 7-10.

How OPTCOMPIND Affects the Query Plan

The OPTCOMPIND setting influences the access plan that the optimizer chooses for single and multitable queries, as the following sections describe.

Single Table Query

For single-table scans, when OPTCOMPIND is set to 0, or when OPTCOMPIND is set to 1 and the current transaction isolation level is Repeatable Read, the optimizer considers the following access plans:

- If an index is available, the optimizer uses it to access the table.
- If no index is available, the optimizer considers scanning the table in physical order.

When OPTCOMPIND is not set in the database server configuration, its value defaults to 2. When OPTCOMPIND is set to 2 or 1 and the current isolation level is not Repeatable Read, the optimizer chooses the least-expensive plan to access the table.

Multitable Query

For join plans, the OPTCOMPIND setting influences the access plan for a specific ordered pair of tables.

If OPTCOMPIND is set to 0, or if it is set to 1 and the current transaction isolation level is Repeatable Read, the optimizer gives preference to the nested-loop join.

If OPTCOMPIND is set to 2, or set to 1 and the transaction isolation level is *not* Repeatable Read, the optimizer chooses the least-expensive query plan from among those previously listed and gives no preference to the nested-loop join.

How Available Memory Affects the Query Plan

The database server constrains the amount of memory that a parallel query can use based on the values of the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters. If the amount of memory available for the query is too low to execute a hash join, the database server uses a nested-loop join instead.

For more information on parallel queries and the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters, refer to Chapter 9, "Parallel Database Query."

Query Plans for Subqueries

The optimizer can change a subquery into a join automatically if the join provides a lower cost. For example, the following sample output of the SET EXPLAIN ON statement shows that the optimizer changes the table in the subquery to be the inner table in a join:

```
QUERY:
------
select col1 from tab1 where exists(
select col1 from tab2 where tab1.col1 = tab2.col1)
Estimated Cost: 144
Estimated # of Rows Returned: 510
1) lsuto.tab1: SEQUENTIAL SCAN
2) lsuto.tab2: INDEX PATH
  (First Row)
    (1) Lower Index Filter: lsuto.tab2.col1 = lsuto.tab1.col1
NESTED LOOP JOIN  (Semi Join)
```

For more information on the SET EXPLAIN ON statement, refer to "How to Display the Query Plan" on page 7-14.

When the optimizer changes a subquery to a join, it can employ several variations of the access plan and the join plan:

- First-row scan

  A first-row scan is a variation of a table scan. As soon as the database server finds one match, the table scan halts.

- Skip-duplicate-index scan

  The skip-duplicate-index scan is a variation of an index scan. The database server does not scan duplicates.

- Semi join

  The semi join is a variation of a nested-loop join. The database server halts the inner-table scan when the first match is found. For more information on a semi join, refer to "Nested-Loop Join" on page 7-5.

An Example of How Query Plans Are Executed

The following SELECT statement 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
```

Assume also that no indexes are on any of the three tables. 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 7-3 shows the query plan, expressed in pseudocode. For information about interpreting query plan information, see "How to Display the Query Plan" on page 7-14.

```
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 7-3
A Query Plan in Pseudocode

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**, looking for a matching **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.

A Join with Column Filters

The presence of a *column filter* changes things. A column filter is a WHERE expression that reduces the number of rows that a table contributes to a join. The following example shows the preceding query with a filter added:

```
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 7-4 displays this sample plan 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 7-4
One of Two Query Plans in Pseudocode

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 7-4 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 7-5 shows an alternative execution plan that 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 7-5
The Alternative Query Plan in Pseudocode

Because the filter is not applied in the first step that Figure 7-5 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 7-4 and Figure 7-5 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.

A Join Using Indexes

The preceding examples do not use indexes or constraints. The presence of indexes and constraints provides the optimizer with options that can greatly improve query-execution time. Figure 7-6 shows the outline of a query plan for the previous query as it might be constructed using indexes.

```
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 7-6
A Query Plan Using Indexes in Pseudocode

The keys in an index are sorted so that when the first matching entry is found, any other rows with identical keys can be read 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 great reduction in effort compared with plans that do not use indexes. An inverse plan, reading **orders** first and looking up rows in the **customer** table by its index, is also feasible by the same reasoning.

Using an index incurs an additional cost over reading the table sequentially. Each entry, or set of entries with the same value, must be located in the index. Then for each entry in the index, a random access must be made to 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, multiple retrievals of orders for a given customer would proceed more rapidly than if the table were ordered randomly.

GLS

When GLS is enabled, indexes that are built on NCHAR or NVARCHAR columns are sorted using a country-specific comparison value. For example, the Spanish double-l character (ll) might be treated as a single unique character instead of a pair of l's. ♦

How the Optimizer Evaluates Query Plans

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 are join pairs that contain the same tables and produce 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 redundant with respect to (B x A).

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.

How to Display the Query Plan

Any user who runs a query can use the SET EXPLAIN ON statement to display the query plan that the optimizer chooses. The user enters the SET EXPLAIN ON statement before the SELECT statement for the query. After the database server executes the SET EXPLAIN ON statement, it writes an explanation of each query plan to a file for subsequent queries that the user enters.

UNIX

On UNIX, the database server writes the output of the SET EXPLAIN ON statement to the **sqexplain.out** file.

If the client application and the database server are on the same computer, the **sqexplain.out** file is stored in your current directory. If you are using a Version 5.x or earlier client application and the **sqexplain.out** file does not appear in the current directory, check your home directory for the file.

When the current database is on another computer, the **sqexplain.out** file is stored in your home directory on the remote host. ♦

WIN NT

On Windows NT, the database server writes the output of the SET EXPLAIN ON statement to the file **%INFORMIXDIR%\sqexpln\*username*.out**. ♦

The SET EXPLAIN output contains the following information:

- The SELECT statement for the query
- An estimate of the query cost in units used by the optimizer to compare plans

  These units represent a relative time for query execution, where 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.
- An estimate for the number of rows that the query is expected to produce
- The order to access the tables during execution
- The access plan by which the database server reads each table

  The following table shows the possible access plans.

Access Plan	Effect
SEQUENTIAL SCAN	Reads rows in sequence
INDEX PATH	Scans one or more indexes
AUTOINDEX PATH	Creates a temporary index
REMOTE PATH	Accesses another distributed database

- The table column or columns that serve as a filter, if any, and whether the filtering occurs through an index
- The join plan for each pair of tables

  The DYNAMIC HASH JOIN section indicates that a hash join is to be used on the preceding join / table pair. It includes a list of the filters used to join the tables together. If DYNAMIC HASH JOIN is followed by the (Build Outer) in the output, the build phase occurs on the first table. Otherwise, the build occurs on the second table, preceding the DYNAMIC HASH JOIN.

The following example shows the SET EXPLAIN output for a simple query and a complex query from the **customer** table:

```
QUERY:
-----------
SELECT fname, lname, company FROM customer

Estimated Cost: 3
Estimated # of Rows Returned: 28

1) joe.customer: SEQUENTIAL SCAN


QUERY:
 ------
SELECT fname, lname, company FROM customer
    WHERE company MATCHES 'Sport*' AND customer_num BETWEEN 110 AND 115
    ORDER BY lname;

Estimated Cost: 4
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

1) joe.customer: INDEX PATH

Filters: joe.customer.company MATCHES 'Sport*'

 (1) Index Keys: customer_num
  Lower Index Filter: joe.customer.customer_num >= 110
  Upper Index Filter: joe.customer.customer_num <= 115
```

Figure 7-7 shows the SET EXPLAIN output for a multiple-table query.

Figure 7-7
Output Produced by 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
```

The SET EXPLAIN output lists the order that the database server accesses the tables and the access plan to read each table. The plan in Figure 7-7 indicates that Dynamic Server is to perform the following actions:

1. Dynamic Server is to read the **orders** table first. Because no filter exists on the **orders** table, Dynamic Server must read all rows. Reading the table in physical order is the least expensive approach.
2. For each row of **orders**, Dynamic Server is to search 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.
3. For each row of **orders** that has a matching **customer_num**, Dynamic Server is to search for a match in the **items** table using the index on **order_num**.

A *key-first scan* is an index scan that uses keys other than those listed as lower and upper index filters. Figure 7-8 shows a sample query using a key-first scan.

Figure 7-8
SET EXPLAIN Output for a Key-First Scan

```
create index idx1 on tab1(c1, c2);

select * from tab1 where (c1 > 0) and ( (c2 = 1) or (c2 = 2))
Estimated Cost: 4
Estimated # of Rows Returned: 1

1) UsErNaMe.tab1: INDEX PATH

    Filters: (UsErNaMe.tab1.c2 = 1 OR UsErNaMe.tab1.c2 = 2)

    (1) Index Keys: c1 c2   (Key-First)  (Serial, fragments: ALL)
        Lower Index Filter: UsErNaMe.tab1.c1 > 0
```

Even though in this example the database server must eventually read the row data to return the query results, it attempts to reduce the number of possible rows by applying additional key filters first. The database server uses the index to apply the additional filter, `c2 = 1 OR c2 = 2`, before it reads the row data.

Factors That Affect the Query Plan

When determining the query plan, the optimizer assigns a cost to each possible plan and then chooses the plan with the lowest cost. Some of the factors that the optimizer uses to determine the cost of each query plan are as follows:

- 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 amount of memory available for the query (specified by the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters)

To calculate the cost of each possible query plan, the optimizer:

- uses a set of statistics that describe the nature and physical characteristics of the table data and indexes.
- examines the query filters.
- examines the indexes that could be used in the plan.

Using Statistics

The accuracy with which the optimizer can assess the execution cost of a query plan depends on the information available to the optimizer. Use the UPDATE STATISTICS statement to maintain simple statistics 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 initializes a statistical profile of a table when the table is created, and the profile is refreshed when you issue the UPDATE STATISTICS statement. The query optimizer does not recalculate the profile for tables automatically. In some cases, gathering the statistics might take longer than executing the query.

To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals. For guidelines on running UPDATE STATISTICS, refer to "Updating Statistics" on page 10-6.

The optimizer uses the following system catalog information as it creates a query plan:

- The number of rows in a table, as of the most recent UPDATE STATISTICS statement
- Whether a column is constrained to be unique
- 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 Distributions" on page 10-7.
- The number of disk pages that contain row data

The optimizer also uses the following system catalog information about indexes:

- The indexes that exist on a table, including the columns they index, whether they are ascending or descending, and whether they are clustered
- The depth of the index structure (a measure of the amount of work that is needed to perform an index lookup)
- The number of disk pages that index entries occupy
- The number of unique entries in an index, which can be used to estimate the number of rows that an equality filter returns
- Second-largest and second-smallest key values in an indexed column

Only the second-largest and second-smallest key values are noted because the extreme values might have a special meaning that is not related to the rest of the data in the column. The database server assumes that key values are distributed evenly between the second largest and second smallest. Only the initial 4 bytes of these keys are stored. If you create a distribution for a column associated with an index, the optimizer uses that distribution when it estimates the number of rows that match a query.

For more information on system catalog tables, refer to the *Informix Guide to SQL: Reference*.

Assessing Filters

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 that is used within the WHERE clause. A conditional expression that is used to select rows is termed a *filter*.

The selectivity is a value between 0 and 1 that indicates the proportion of rows within the table that the filter can pass. A very selective filter, one that passes few rows, has a selectivity near 0, and a filter that passes almost all rows has a selectivity near 1. For guidelines on filters, see “Improving Filter Selectivity” on page 10-5.

The optimizer can use data distributions to calculate selectivities for the filters in a query. However, in the absence of data distributions, the database server calculates selectivities for filters of different types based on table indexes. The following table lists some of the selectivities that the optimizer assigns to filters of different types. Selectivities calculated using data distributions are even more accurate than the ones that this table shows.

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
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*))

(1 of 2)

Filter Expression	Selectivity (F)
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)

Assessing Indexes

The optimizer notes whether an index 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. If the values contained in the index are all that is required, the rows are not read. It is faster to omit the page lookups for data pages whenever values can be read directly from the 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

  The database server can locate relevant rows in the table by first finding the row in an appropriate index. If an appropriate index is not available, the database server must scan each table in its entirety.

- 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 within 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.

Time Costs of a Query

This section explains the response-time effects of actions that the database server performs as it processes a query.

Many of the costs described cannot be reduced by adjusting the construction of the query. A few can be, however. The following costs can be reduced by optimal query construction and appropriate indexes:

- Sort time
- Data mismatches
- In-place ALTER TABLE
- Index lookups

For information about how to optimize specific queries, see Chapter 10, “Improving Individual Query Performance.”

Memory Activity Costs

The database server can process only data in memory. It must read rows into memory to evaluate those rows against the filters of a query. Once rows that satisfy those filters are found, the database server prepares an output row in memory by assembling the selected columns.

Most of these activities are performed very quickly. Depending on the computer and its work load, 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 and 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 and stands for the work to copy or compare one sort key. A 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 can involve writing information temporarily to disk if there is a large amount of data to sort. You can direct the disk writes to occur in the operating-system file space or in a dbspace that the database server manages. For details, refer to “Dbspaces for Temporary Tables and Sort Files” on page 3-50.

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. You can 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 can be held in memory. If there are no filters, then N_{fr}/N_{rp} is equivalent to *p*.

When all the keys can be held in memory, *m*=1 and the disk work is equivalent to *p*. In other words, the rows are read and sorted in memory.

For moderate- to large-sized 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:

$$W_d = \quad +p(2 * (N_{fr}/N_{rp}))$$

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 pages.

The actual cost of reading a page is variable and hard to predict. It is a combination of the following factors.

Factor	Effect of Factor
Buffering	If the needed page is in a page buffer already, the cost to read is nearly zero.
Contention	If two or more applications require access to the disk hardware, I/O requests can be delayed.
Seek time	The slowest thing that a disk does is to *seek*; that is, to move 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 zero to nearly a second.
Latency	The transfer cannot start until the beginning of the page rotates under the access arm. This *latency*, or rotational delay, depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from zero 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 zero 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. When one page is exhausted, the page for the next set of rows must be read in. To make sure that the next page is ready in memory, use the read-ahead configuration parameters described in "RA_PAGES and RA_THRESHOLD" on page 3-60.

When you use unbuffered devices for dbspaces, and the table is organized properly, the disk pages of consecutive rows are placed in consecutive locations on the disk. This arrangement allows the access arm to move very little when reading 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 reading a table nonsequentially than when reading that same table 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 Look-Up 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 look-up works down from the root page to a leaf page. 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. However, if there are many duplicate rows per 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

For certain conditions, the database server uses an in-place alter algorithm to modify each row when you execute an ALTER TABLE statement (rather than during the alter table operation). After the alter table operation, the database server inserts rows using the latest definition.

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 4-40.

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 when the complexity of the view definition causes a temporary table to be materialized to process the query. For example, you can create a view with a union to combine results from several SELECT statements.

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 handle the complexity. The end user can just 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 expected because the database server creates a fragmented temporary table for the view before it executes the query.

To determine if you have a query that must build a temporary table to materialize the view, execute the SET EXPLAIN statement. If you see `Temp Table For View` in the SET EXPLAIN output 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.

As an example, in the **stores7** database, the **state** table that relates abbreviations to names of states has a total size less than 1,000 bytes; it fits in no more than two pages. This table can be included in any query at little cost. No matter how this table is used, it costs no more than two disk accesses to retrieve this table from disk the first time that it is required.

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 SET EXPLAIN output shows the query in its adjusted format. This data conversion has no noticeable overhead.

The additional costs of a data mismatch are most severe 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 quotes:

```
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 SET EXPLAIN output 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 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*.

Network-Access Costs

Moving data over a network imposes delays in addition to those you encounter with direct disk access. Network delays can occur when the application sends a query or update request across the network to a database server on another computer. Although the database server performs the query on the remote host computer, that database server returns the output to the application over the network.

Data sent over a network consists of command messages and buffer-sized blocks of row data. Although the details can differ depending on the network and the computers, the database server network activity follows a simple model in which one computer, the *client*, sends a request to another computer, the *server*. The server replies with a block of data from a table.

Whenever data is exchanged over a network, delays are inevitable in the following situations:

- When the network is busy, the client must wait its turn to transmit. Such delays are usually less than a millisecond. However, on a heavily loaded network, these delays can increase exponentially to tenths of seconds and more.
- When the server is handling requests from more than one client, requests might be queued for a time that can range from milliseconds to seconds.
- When the server acts on the request, it incurs the time costs of disk access and in-memory operations that the preceding sections describe.

Transmission of the response is also subject to network delays.

Network access time is extremely variable. In the best case, when neither the network nor the server is busy, transmission and queueing delays are insignificant, and the server sends a row almost as quickly as a local database server might. Furthermore, when the client asks for a second row, the page is likely to be in the page buffers of the server.

Unfortunately, as network load increases, all these factors tend to worsen at the same time. Transmission delays rise in both directions, which increases the queue at the server. The delay between requests decreases the likelihood of a page remaining in the page buffer of the responder. Thus, network-access costs can change suddenly and quite dramatically.

The optimizer that the database server uses assumes that access to a row over the network takes longer than access to a row in a local database. This estimate includes the cost of retrieving the row from disk and transmitting it across the network.

SQL Within Stored Procedures

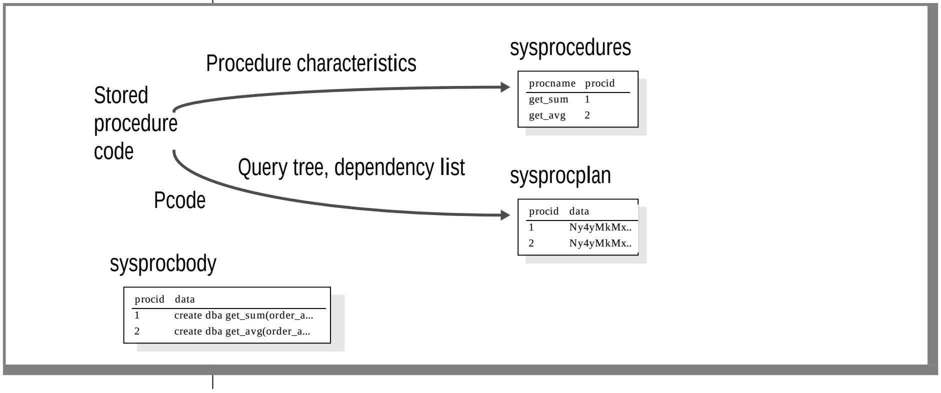
The following sections contain information about how and when the database server optimizes and executes SQL within a stored procedure.

When SQL Is Optimized

If a stored procedure contains SQL statements, at some point the query optimizer evaluates the possible query plans for SQL in the stored procedure and selects the query plan with the lowest cost. The database server puts the selected query plan for each SQL statement into an execution plan for the stored procedure.

When you create a stored procedure with the CREATE PROCEDURE statement, the database server attempts to optimize the SQL statements within the stored procedure at that time. If the tables cannot be examined at compile time (they might not exist or might not be available), the creation does not fail. In this case, the database server optimizes the SQL statements the first time that the stored procedure executes. The database server stores the optimized execution plan in the **sysprocplan** system catalog table for use by other processes. In addition, the database server stores information about the stored procedure (such as procedure name and owner) in the **sysprocedures** system catalog table and an ASCII version of the stored procedure in the **sysprocbody** system catalog table.

Figure 7-9 summarizes the information that the database server stores in system catalog tables during the compilation process.



sysprocedures

procname	procid
get_sum	1
get_avg	2

sysprocplan

procid	data
1	Ny4yMkMx..
2	Ny4yMkMx..

sysprocbody

procid	data
1	create dba get_sum(order_a...
2	create dba get_avg(order_a...

Figure 7-9
SPL Information Stored in System Catalog Tables

Automatic Optimization

The database server uses the dependency list to keep track of changes that would cause reoptimization the next time that a Stored Procedure Language (SPL) routine executes. The database server reoptimizes an SQL statement the next time that an SPL routine executes after one of the following situations:

- Execution of any data definition language (DDL) statement (such as ALTER TABLE, DROP INDEX, CREATE INDEX) that might alter the query plan
- Alteration of a table that is linked to another table with a referential constraint (in either direction)
- Execution of UPDATE STATISTICS FOR TABLE for any table involved in the query

  The UPDATE STATISTICS FOR TABLE statement changes the version number of the specified table in **systables.**

The database server updates the **sysprocplan** system catalog table with the reoptimized execution plan.

Optimization Levels for SQL in Stored Procedures

The current optimization level set in an SPL routine affects how the SPL routine is optimized.

The algorithm that a SET OPTIMIZATION HIGH statement invokes is a sophisticated, cost-based strategy that examines all reasonable query plans and selects the best overall alternative. For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.

The alternative algorithm that a SET OPTIMIZATION LOW statement invokes eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization. However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

For stored procedures that remain unchanged or change only slightly and that contain complex SELECT statements, you might want to set the SET OPTIMIZATION statement to `HIGH` when you create the stored procedure. This optimization level stores the best query plans for the stored procedure. Then set optimization to `LOW` before you execute the stored procedure. The stored procedure then uses the optimal query plans and runs at the more cost-effective rate if reoptimization occurs.

How a Stored Procedure Is Executed

When the database server executes a stored procedure with the EXECUTE PROCEDURE statement, the SPL CALL statement, or within 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 retrieves the query plan from the database and executes the statement. If the query plan has not been created, the database server optimizes the SQL statement before executing it.
- When the database server reaches the end of the stored procedure 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 stored procedure execute is complete.

Chapter

Optimizer Directives

8

This chapter describes how to use directives to improve query performance.

This chapter discusses the following topics:

- Purpose of directives
- How you use directives
- Configuration parameters and environment variables that affect directives
- Directives and stored procedures
- Guidelines for using directives

Optimizer Directives

Optimizer directives are comments in a SELECT statement that instruct the query optimizer how to execute a query. You can also place directives in UPDATE and DELETE statements, instructing the optimizer how to access the data. Optimizer directives can either be explicit directions (for example, “use this index,” or “access this table first”), or they can eliminate possible query plans (for example, “do not read this table sequentially,” or “do not perform a nested-loop join”).

Why Use Optimizer Directives?

You can use optimizer directives when the optimizer does not choose the best query plan to perform a query. The result of a poor query plan can be reduced performance.

Before you decide when to use optimizer directives, you should understand what makes a good query plan.

Although the optimizer creates a query plan based on costs of using different table access paths, join orders, and join plans, it generally chooses a query plan that follows these guidelines:

- Do not use an index when the database server must read a large portion of the table. For example, the following query might read most of the customer table:

  ```
  SELECT * FROM customer WHERE STATE <> "ALASKA";
  ```

  Assuming the customers are evenly spread among all 50 states, you might estimate that the database server must read 98 percent of the table. It is more efficient to read the table sequentially than it is to traverse an index (and subsequently the data pages) when the database server must read most of the rows.

- When you choose between indexes to access a table, use an index that can rule out the most rows. For example, consider the following query:

  ```
  SELECT * FROM customer
  WHERE state = "NEW YORK" AND order_date = "01/20/97"
  ```

  Assuming that 200,000 customers live in New York and only 1000 customers ordered on any one day, the optimizer most likely chooses an index on **order_date** rather than an index on **state** to perform the query.

- Place small tables or tables with restrictive filters early in the query plan. For example, consider the following query:

  ```
  SELECT * FROM customer, orders
      WHERE customer.customer_num = orders.customer_num
          AND
      customer.state = "NEVADA";
  ```

  In this example, by reading the customer table first, you can rule out most of the rows by applying the filter that chooses all rows in which `state = "NEVADA"`.

  By ruling out rows in the customer table, the database server does not read as many rows in the **orders** table (which might be significantly larger than the **customer** table).

- Choose a hash join when neither column in the join filter has an index.

  In the previous example, if **customer.customer_num** and **orders**.**customer_num** are not indexed, a hash join is probably the best join plan.

- Choose nested-loop joins if:
  - the number of rows retrieved from the outer table after the database server applies any table filters is small, and the inner table has an index that can be used to perform the join.
  - the index on the outermost table can be used to return rows in the order of the ORDER BY clause, eliminating the need for a sort.

Before You Use Directives

In most cases, the optimizer chooses the fastest query plan. To assist the optimizer, make sure that you perform the following tasks:

- Run UPDATE STATISTICS.

  Without accurate statistics, the optimizer cannot choose the appropriate query plan. Run UPDATE STATISTICS any time that the data in the tables changes significantly (many new rows are added, updated, or deleted).

- Create distributions.

  One of the first things that you should try when you attempt to improve a slow query is to create distributions on columns involved in a query. Distributions provide the most accurate information to the optimizer about the nature of the data in the table. Run UPDATE STATISTICS HIGH on columns involved in the query filters to see if performance improves.

In some cases, the query optimizer does not choose the best query plan because of the complexity of the query or because (even with distributions) it does not have enough information about the nature of the data. In these cases, you can attempt to improve performance for a particular query by using directives.

Types of Directives

Include directives in the SQL statement as a comment that occurs immediately after the SELECT keyword. The first character in a directive is always a plus (+) sign. In the following query, the ORDERED directive specifies that the tables should be joined in the same order as they are specified in the FROM clause. The AVOID_FULL directive specifies that the optimizer should discard any plans that include a full table scan on the listed table (**employee**).

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM employee e, department d
WHERE e.dept_no = d. dept_no AND e.salary > 50000;
```

For a complete syntax description for directives, refer to the *Informix Guide to SQL: Syntax*.

To influence the query plan choice the optimizer makes, you can alter four aspects of a query: the access plan, the join order, the join plan, and the optimization goal. The following pages explain these plans in detail.

Access Plan Directives

The access plan is the method that the database server uses to access a table. The database server can either read the table sequentially (full table scan) or use any one of the indexes on the table. The following directives influence the access plan:

- INDEX

  Use the index specified o access the table. If the directive lists more than one index, the optimizer chooses the index that yields the least cost.

- AVOID_INDEX

  Do not use any of the indexes listed. You can use this directive with the AVOID_FULL directive.

- FULL

  Perform a full table scan.
- AVOID_FULL

  Do not perform a full table scan on the listed table. You can use this directive with the AVOID_INDEX directive.

In some cases, forcing an access method can change the join method that the optimizer chooses. For example, if you exclude the use of an index with the AVOID_INDEX directive, the optimizer might choose a hash join instead of a nested-loop join.

Join Order Directives

The join order directive ORDERED forces the optimizer to join tables in the order that the SELECT statement lists them.

By specifying the join order, you might affect more than just how tables are joined. For example, consider the following query:

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM employee e, department d
WHERE e.dept_no = d.dept_no AND e.salary > 5000
```

In this example, the optimizer chooses to join the tables with a hash join. However, if you arrange the order so that the second table is **employee** (and must be accessed by an index), the hash join is not feasible.

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM department d, employee e
WHERE e.dept_no = d.dept_no AND e.salary > 5000;
```

The optimizer chooses a nested-loop join in this case.

Join Order When You Use Views

Two cases can affect join order when you use views:

- The ORDERED directive is inside the view.

  The ORDERED directive inside a view affects the join order of only the tables inside the view. The tables in the view must be joined contiguously. Consider the following view and query:

  ```
  CREATE VIEW emp_job_view as
           SELECT {+ORDERED}
           emp.job_num, job.job_name
           FROM emp, job
           WHERE emp.job_num = job.job_num;

  SELECT * from dept, emp_job_view,  project
           WHERE dept.dept_no = project.dept_num
           AND emp_job_view.job_num = project.job_num;
  ```

  The ORDERED directive specifies that the **emp** table come before the **job** table. The directive does not affect the order of the **dept** and **project** table. Therefore, all possible join orders are as follows:

  - **emp**,**job**,**dept**,**project**
  - **emp**,**job**,**project**,**dept**
  - **project**,**emp**,**job**,**dept**
  - **dept**,**emp**,**job**,**project**
  - **dept**,**project**,**emp**,**job**
  - **project**,**dept**,**emp**,**job**

- The ORDERED directives in a query that contains a view.

  If an ORDERED directive appears in a query that contains a view, the join order of the tables in the query are the same as they are listed in the SELECT statement. The tables within the view are joined as they are listed within the view.

  In the following query, the join order is **dept**, **project**, **emp**, **job**:

  ```
  CREATE VIEW emp_job_view AS
           SELECT
           emp.job_num, job.job_name
           FROM emp, job
           WHERE emp.job_num = job.job_num;
  SELECT {+ORDERED}
           * FROM dept, project, emp_job_view
           WHERE dept.dept_no = project.dept_num
           AND emp_job_view.job_num = project.job_num;
  ```

  An exception to this rule is when the view is one that cannot be folded into the query, as in the following example:

  ```
  CREATE VIEW emp_job_view2 AS
       SELECT DISTINCT
       emp.job_num, job.job_name
       FROM emp,job
       WHERE emp.job_num = job.job_num;
  ```

  In this example, the database server executes the query and puts the result into a temporary table. The order of tables in the previous query is then **dept**, **project**, ***temp_table***.

Join Plan Directives

The join plan directives influence how the database server joins two tables in a query.

The following directives influence the join plan between two tables:

- USE_NL

  Use the listed tables as the inner table in a nested-loop join.

- USE_HASH

  Access the listed tables with a hash join. You can also choose whether the table will be used to create the hash table or to probe the hash table.

- AVOID_NL

  Do not use the listed tables as the inner table in a nested-loop join. A table listed with this directive can still participate in a nested-loop join as an outer table.
- AVOID_HASH

  Do not access the listed tables with a hash join. Optionally, you can allow a hash join but restrict the table from being the one that is probed or the table from which the hash table is built.

Optimization Goal Directives

In some queries, you might want to find only the first few rows in the result of a query (for example, an ESQL/C program opens a cursor for the query and performs a FETCH to find only the first row). Or you might know that all rows must be accessed and returned from the query. You can use the optimization goal directives to optimize the query for either one of these cases:

- FIRST_ROWS

  Choose a plan that optimizes the process of finding only the first row that satisfies the query.
- ALL_ROWS

  Choose a plan the optimizes the process of finding all rows (the default behavior) that satisfy the query.

If you use the FIRST_ROWS directive, the optimizer might abandon a query plan that contains activities that are time-consuming upfront. For example, a hash join might take too much time to create the hash table. If only a few rows must be returned, the optimizer might choose a nested-loop join instead.

In the following example, assume that the database has an index on **employee.dept_no** but not on **department.dept_no**. Without directives, the optimizer chooses a hash join.

```
SELECT *
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

However, with the FIRST_ROWS directive, the optimizer chooses a nested-loop join because of the high initial overhead required to create the hash table.

```
SELECT {+first_rows(1)} *
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

An Example with Directives

The following example shows how directives can alter the query plan.

Suppose you have the following query:

```
SELECT * FROM emp,job,dept
WHERE emp.location = 10
    AND emp.jobno = job.jobno
    AND emp.deptno = dept.deptno
    AND dept.location = "DENVER";
```

Assume that the following indexes exist:

```
ix1: emp(empno,jobno,deptno,location)
ix2: job(jobno)
ix3: dept(location)
```

You run the query with SET EXPLAIN ON to display the query path that the optimizer uses.

```
QUERY:
------
SELECT * FROM emp,job,dept
WHERE emp.location = "DENVER"
    AND emp.jobno = job.jobno
    AND emp.deptno = dept.deptno
    AND dept.location = "DENVER"

Estimated Cost: 5
Estimated # of Rows Returned: 1

1) informix.emp: INDEX PATH

    Filters: informix.emp.location = 'DENVER'

    (1) Index Keys: empno jobno deptno location   (Key-Only)

2) informix.dept: INDEX PATH

    Filters: informix.dept.deptno = informix.emp.deptno

    (1) Index Keys: location
        Lower Index Filter: informix.dept.location = 'DENVER'
NESTED LOOP JOIN

3) informix.job: INDEX PATH

    (1) Index Keys: jobno   (Key-Only)
        Lower Index Filter: informix.job.jobno = informix.emp.jobno
NESTED LOOP JOIN
```

The diagram in Figure 8-1 shows a possible query plan for this query.

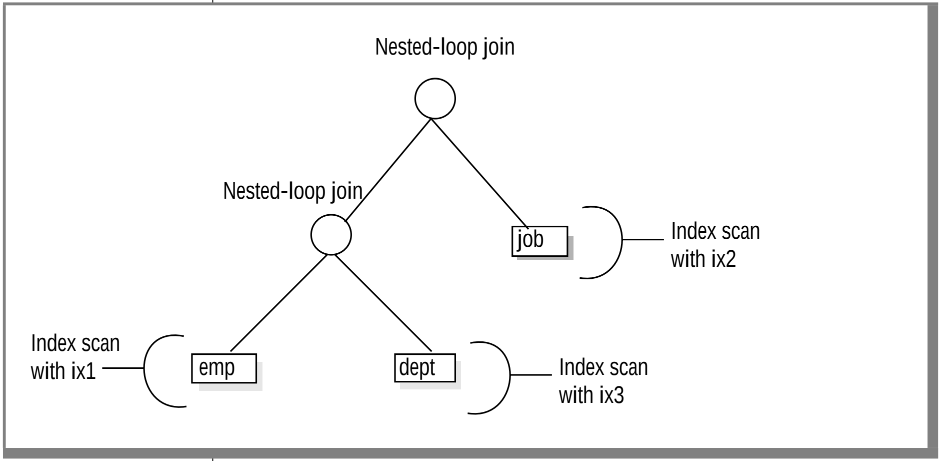


Figure 8-1
Possible Query Plan Without Directives

Perhaps you are concerned that using a nested-loop join might not be the fastest method to execute this query. You also think that the join order is not optimal. You can force the optimizer to choose a hash join and order the tables in the query plan according to their order in the query, so the optimizer uses the query plan shown in Figure 8-2.

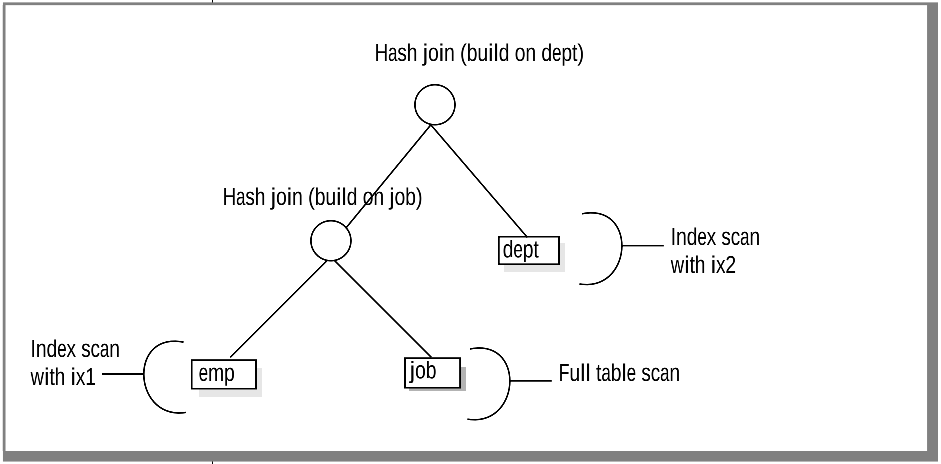


Figure 8-2
Possible Query Plan with Directives

To force the optimizer to choose the query plan that uses hash joins and the order of tables shown in the query, you use the directives that the following SET EXPLAIN output shows:

```
QUERY:
------
SELECT {+ORDERED,
    INDEX(emp ix1),
    FULL(job),
    USE_HASH(job /BUILD),
    USE_HASH(dept /BUILD),
    INDEX(dept ix3)}
    * FROM emp,job,dept
    WHERE emp.location = 1
    AND emp.jobno = job.jobno
    AND emp.deptno = dept.deptno
    AND dept.location = "DENVER"

DIRECTIVES FOLLOWED:
ORDERED
INDEX ( emp ix1 )
FULL ( job )
USE_HASH ( job/BUILD )
USE_HASH ( dept/BUILD )
INDEX ( dept ix3 )
DIRECTIVES NOT FOLLOWED:

Estimated Cost: 7
Estimated # of Rows Returned: 1

1) informix.emp: INDEX PATH

    Filters: informix.emp.location = 'DENVER'

    (1) Index Keys: empno jobno deptno location   (Key-Only)

2) informix.job: SEQUENTIAL SCAN


DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.emp.jobno = informix.job.jobno

3) informix.dept: INDEX PATH

    (1) Index Keys: location
        Lower Index Filter: informix.dept.location = 'DENVER'

DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.emp.deptno = informix.dept.deptno
```

Directives Configuration Parameters and Environment Variables

You can use the DIRECTIVES configuration parameter to turn on or off all directives that the database server encounters. If DIRECTIVES is 1 (the default), the optimizer follows all directives. If DIRECTIVES is 0, the optimizer ignores all directives.

You can override the setting of DIRECTIVES with the **IFX_DIRECTIVES** environment variable. If **IFX_DIRECTIVES** is set to 1 or ON, the optimizer follows directives for any SQL executed by the client session. If **IFX_DIRECTIVES** is 0 or OFF, the optimizer ignores directives for any SQL in the client session.

Any directives in an SQL statement take precedence over the join plan forced by the OPTCOMPIND configuration parameter. For example, if a query includes the USE_HASH directive and OPTCOMPIND is set to 0 (nested-loop joins preferred over hash joins), the optimizer uses a hash join.

Directives and Stored Procedures

Directives operate differently for a query in a stored procedure because a SELECT statement in a stored procedure does not necessarily get optimized immediately before the database server executes it. The optimizer creates a query plan for a SELECT statement in a stored procedure when the database server creates a stored procedure or during the execution of some versions of the UPDATE STATISTICS statement.

The optimizer reads and applies directives at the time that it creates the query plan. Because it stores the query plan in a system catalog table, the SELECT statement is not reoptimized when it is executed. Therefore, settings of **IFX_DIRECTIVES** and DIRECTIVES affect SELECT statements inside a stored procedure when they are set before the CREATE PROCEDURE statement, before the UPDATE STATISTICS statements that cause stored procedure SQL to be optimized, or during certain circumstances when SELECT statements have variables supplied at runtime.

Guidelines for Using Directives

Consider the following guidelines when you use directives:

- Examine the effectiveness of a particular directive frequently. Imagine a query in a production program with several directives that force an optimal query plan. Some days later, users add, update, or delete a large number of rows, which changes the nature of the data so much that the once optimal query plan is no longer effective. This is but one example of how you must use directives with care.
- Use negative directives (AVOID_NL, AVOID_FULL, and so on) whenever possible. When you exclude a behavior that degrades performance, you rely on the optimizer to use the next best choice, rather than attempt to force a path that might not be optimal.

Chapter

Parallel Database Query

9

P*arallel database query* (PDQ) is an Informix database server feature that can improve performance dramatically when the database server processes queries initiated by decision-support applications. PDQ features allow the database server to distribute the work for 3eone aspect of a query among several processors. For example, if a query requires an aggregation, the database server can distribute the work for the aggregation among several processors. PDQ also includes tools for resource management.

Another database server feature, *table fragmentation*, allows you to store the parts of a table on different disks. PDQ delivers maximum performance benefits when the data that is being queried is in fragmented tables. “Planning a Fragmentation Strategy” on page 6-4 describes how to use fragmentation for maximum performance.

This chapter discusses the parameters and strategies that you use to manage resources for PDQ. This chapter covers the following topics:

- How the optimizer structures a PDQ query
- The Memory Grant Manager
- Allocating Dynamic Server resources for PDQ
- Balancing resource requirements for on-line transaction processing (OLTP) and decision-support applications
- Monitoring PDQ resources

How the Optimizer Structures a PDQ Query

Each decision-support query has a primary thread. The database server can start additional threads to perform tasks for the query (for example, scans and sorts). Depending on the number of tables or fragments that a query must search and the resources that are available for a decision-support query, the database server assigns different components of a query to different threads. The database server initiates these PDQ threads, which are listed as *secondary threads* in the SET EXPLAIN output.

Secondary threads are further 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 along 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 along to a sort thread. When doing so, the join thread is considered a producer, and the sort thread is considered a consumer.

Several producers can supply data to a single consumer. When this occurs, Dynamic Server sets up an internal mechanism, called an *exchange*, that synchronizes the transfer of data from those producers to the consumer. For instance, if a fragmented table is to be sorted, the optimizer typically calls for a separate scan thread for each fragment. Because of different I/O characteristics, the scan threads can be expected to complete at different times. An exchange is used to funnel the data produced by the various scan threads into one or more sort threads with a minimum amount of buffering. Depending on the complexity of the query, the optimizer might call for a multi-layered hierarchy of producers, exchanges, and consumers. Generally speaking, consumer threads work in parallel with producer threads so that the amount of intermediate buffering performed by the exchanges remains negligible.

These threads and exchanges are created automatically and transparently. They terminate automatically as they complete processing for a given query. Dynamic Server creates new threads and exchanges as needed for subsequent queries.

The Memory Grant Manager

The Memory Grant Manager (MGM) is an Dynamic Server component that coordinates the use of memory, CPU virtual processors (VPs), disk I/O, and scan threads among decision-support queries. MGM uses the DS_MAX_QUERIES, DS_TOTAL_MEMORY, DS_MAX_SCANS, and MAX_PDQPRIORITY configuration parameters to determine the quantity of these PDQ resources that can be granted to a decision-support query. For more information about these configuration parameters, refer to Chapter 3, "Configuration Impacts on Performance."

The MGM dynamically allocates the following resources for decision-support queries:

- The number of scan threads started for each decision-support query
- The number of threads that can be started for each query
- The amount of memory in the virtual portion of Dynamic Server shared memory that the query can reserve

When your Dynamic Server system has heavy OLTP use and you find performance is degrading, you can use the MGM facilities to limit the resources committed to decision-support queries. During off-peak hours, you can designate a larger proportion of the resources to parallel processing, which achieves higher throughput for decision-support queries.

Memory is granted to a query by the MGM for such activities as sorts, hash joins and processing of GROUP BY clauses. The amount of memory used by decision-support queries cannot exceed DS_TOTAL_MEMORY.

The MGM grants memory to queries in *quantum* increments. A quantum is calculated using the following formula:

```
quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

For example, if DS_TOTAL_MEMORY is 12 megabytes and DS_MAX_QUERIES is 4, then the quantum is 12/4 = 3 megabytes. Thus, with these values in effect, a quantum of memory equals 3 megabytes. In general, memory is allocated more efficiently when quanta are smaller. You can often improve performance of concurrent queries by increasing DS_MAX_QUERIES to reduce the size of a quantum of memory.

You can monitor resources allocated by the MGM by running the **onstat -g mgm** command. This command displays only the amount of memory that is currently used; it does not display the amount of memory that has been granted. For more information about this command, refer to your *Administrator's Guide*.

The MGM also grants a maximum number of scan threads per query based on the values of the DS_MAX_SCANS and the DS_MAX_QUERIES parameters.

The maximum number of scan threads per query is given by the following formula:

```
scan_threads = min (nfrags, DS_MAX_SCANS * (pdqpriority / 100)
                * (MAX_PDQPRIORITY / 100))
```

nfrags	is the number of fragments in the table with the largest number of fragments.
pdqpriority	is the PDQ priority value that is set by either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

For more information about any of these Dynamic Server configuration parameters, refer to Chapter 3, "Configuration Impacts on Performance."

The **PDQPRIORITY** environment variable and the SQL statement SET PDQPRIORITY request a percentage of PDQ resources for a query.

You can use the MAX_PDQPRIORITY configuration parameter to limit the percentage of the requested resources that a query can obtain and to limit the impact of decision-support queries on OLTP processing. For more information, refer to "Limiting the Priority of DSS Queries" on page 9-7.

Allocating Resources for PDQ Queries

When Dynamic Server uses PDQ to perform a query in parallel, it puts a heavy load on the operating system. In particular, PDQ exploits the following resources:

- Memory
- CPU VPs
- Disk I/O (to fragmented tables, temporary table space)
- Scan threads

When you configure Dynamic Server, consider how the use of PDQ affects OLTP users, users of decision-support applications, and users of other applications.

You can control how Dynamic Server uses resources in the following ways:

- Limit the priority of PDQ queries.
- Adjust the amount of memory.
- Limit the number of scan threads.
- Limit the number of concurrent queries.

Limiting the Priority of DSS Queries

The default value for the PDQ priority of individual applications is 0, which means that PDQ processing is not used. Dynamic Server uses this value unless it is overridden by one of the following actions:

- The user sets the **PDQPRIORITY** environment variable.
- The application uses the SET PDQPRIORITY statement.

The **PDQPRIORITY** environment variable and the MAX_PDQPRIORITY configuration parameter work together to control the amount of resources to allocate for parallel processing. Setting these configuration parameters correctly is critical for the effective operation of PDQ.

The MAX_PDQPRIORITY configuration parameter allows the Dynamic Server administrator to limit the parallel processing resources consumed by DSS queries. Thus, the **PDQPRIORITY** environment variable sets a *reasonable* or *recommended* priority value, and MAX_PDQPRIORITY limits the resources that an application can claim.

The MAX_PDQPRIORITY configuration parameter specifies the maximum percentage of the requested resources that a query can obtain. For instance, if **PDQPRIORITY** is 80 and MAX_PDQPRIORITY is 50, each active query receives an amount of memory equal to 40 percent of DS_TOTAL_MEMORY, rounded down to the nearest quantum. In this example, MAX_PDQPRIORITY effectively limits the number of concurrent decision-support queries to two. Subsequent queries must wait for earlier queries to finish before they acquire the resources that they need to run.

An application or user can use the DEFAULT tag of the SET PDQPRIORITY statement to use the value for PDQ priority if the value has been set by the **PDQPRIORITY** environment variable. DEFAULT is the symbolic equivalent of a -1 value for PDQ priority.

You can use the **onmode** command-line utility to change the values of the following configuration parameters temporarily:

- Use **onmode -M** to change the value of DS_TOTAL_MEMORY.
- Use **onmode -Q** to change the value of DS_MAX_QUERIES.
- Use **onmode -D** to change the value of MAX_PDQPRIORITY.
- Use **onmode -S** to change the value of DS_MAX_SCANS.

These changes remain in effect only as long as Dynamic Server remains up and running. When Dynamic Server is initialized, it uses the values listed in the ONCONFIG file.

For more information about the preceding parameters, refer to Chapter 3. For more information about **onmode**, refer to your *Administrator's Guide*.

If you must change the values of the decision-support parameters on a regular basis (for example, to set MAX_PDQPRIORITY to 100 each night for processing reports), you can set the values using a scheduled operating-system job. For information about creating scheduled jobs, refer to your operating-system manuals.

To obtain the best performance from Dynamic Server, choose values for the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter, observe the resulting behavior, and then adjust the values for these parameters. No well-defined rules exist for choosing these environment variable and parameter values. The following sections discuss strategies for setting **PDQPRIORITY** and MAX_PDQPRIORITY for specific needs.

Limiting the Value of PDQ Priority

The MAX_PDQPRIORITY configuration parameter limits the PDQ priority that Dynamic Server grants when users either set the **PDQPRIORITY** environment variable or issue the SET PDQPRIORITY statement before they issue a query. When an application or an end user attempts to set a PDQ priority, the priority that is granted is multiplied by the value specified in MAX_PDQPRIORITY.

Set the value of MAX_PDQPRIORITY lower when you want to allocate more resources to OLTP processing. Set the value higher when you want to allocate more resources to decision-support processing. The possible range of values is `0` to `100`. This range represents the percent of resources that you can allocate to decision-support processing.

Maximizing OLTP Throughput

At times, you might want to allocate resources to maximize the throughput for individual OLTP queries rather than for decision-support queries. In this case, set MAX_ PDQPRIORITY to `0`, which limits the value of PDQ priority to `OFF`. A PDQ priority value of `OFF` does not prevent decision-support queries from running. Instead, it causes the queries to run without parallelization. In this configuration, response times for decision-support queries might be very slow.

Conserving Resources

If applications make little use of queries that require parallel sorts and parallel joins, consider using the `LOW` setting for PDQ priority.

If Dynamic Server is operating in a multiuser environment, you might set MAX_PDQPRIORITY to `1` to increase interquery performance at the cost of some intraquery parallelism. A trade-off exists between these two different types of parallelism because they compete for the same resources. As a compromise, you might set MAX_PDQPRIORITY to some intermediate value (perhaps `20` or `30`) and set **PDQPRIORITY** to `LOW`. The environment variable sets the default behavior to `LOW`, but the MAX_PDQPRIORITY configuration parameter allows individual applications to request more resources with the SET PDQPRIORITY statement.

Allowing Maximum Use of Parallelism

Set **PDQPRIORITY** and MAX_PDQPRIORITY to `100` if you want Dynamic Server to assign as many resources as possible to parallel processing. This setting is appropriate for times when parallel processing does not interfere with OLTP processing.

Determining the Level of Parallelism

You can use different numeric settings for **PDQPRIORITY** to experiment with the effects of parallelism on a single application. For information on how to monitor parallel execution, refer to "Monitoring PDQ Resources" on page 9-17.

Limits on Parallelism Associated with PDQPRIORITY

Dynamic Server reduces the PDQ priority of queries that contain outer joins to `LOW` (if set to a higher value) for the duration of the query. If a subquery or a view contains outer joins, Dynamic Server lowers the PDQ priority only of that subquery or view, not of the parent query or of any other subquery.

Dynamic Server lowers the PDQ priority of queries that require access to a remote database (same or different Dynamic Server instance) to `LOW` if you set it to a higher value. In that case, all local scans are parallel, but all local joins and remote accesses are nonparallel.

Using Stored Procedures

Dynamic Server freezes the PDQ priority that is used to optimize SQL statements within procedures at the time of procedure creation or the last manual recompilation with the UPDATE STATISTICS statement. You can change the client value of **PDQPRIORITY** by embedding the SET PDQPRIORITY statement within the body of your procedure.

The PDQ priority value that Dynamic Server uses to optimize or reoptimize an SQL statement is the value that was set by a SET PDQPRIORITY statement, which must have been executed within the same procedure. If no such statement has been executed, the value that was in effect when the procedure was last compiled or created is used.

The PDQ priority value currently in effect outside a procedure is ignored within a procedure when it is executing.

Informix suggests that you turn PDQ priority off when you enter a procedure and then turn it on again for specific statements. You can avoid tying up large amounts of memory for the procedure, and you can make sure that the crucial parts of the procedure use the appropriate PDQ priority, as the following example illustrates:

```
CREATE PROCEDURE my_proc (a INT, b INT, c INT)
    Returning INT, INT, INT;
SET PDQPRIORITY 0;
...
SET PDQPRIORITY 85;
SELECT ..... (big complicated SELECT statement)
SET PDQPRIORITY 0;
...
;
```

Adjusting the Amount of 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 - os_mem - rsdnt_mem - (128K * users)
                - other_mem
```

p_mem	represents the total physical memory that is available on the host computer.
os_mem	represents the size of the operating system, including the buffer cache.
resdnt_mem	represents the size of Informix resident shared memory.
users	is the number of expected users (connections) specified in the NETTYPE configuration parameter.
other_mem	is the size of memory used for other (non-Informix) applications.

The value for DS_TOTAL_MEMORY that is derived from this formula serves only as a starting point. To arrive at a value that makes sense for your configuration, you must monitor paging and swapping. (Use the tools provided with your operating system to monitor paging and swapping.) When paging increases, decrease the value of DS_TOTAL_MEMORY so that processing the OLTP workload can proceed.

The amount of memory that is granted to a single PDQ query is influenced by many system factors but, in general, the amount of memory granted to a single PDQ query is proportional to the following formula:

```
memory_grant_basis = (DS_TOTAL_MEMORY/DS_MAX_QUERIES) *
                     (PDQPRIORITY / 100) *
                     (MAX_PDQPRIORITY / 100)
```

Limiting the Number of Concurrent Scans

Dynamic Server apportions some number of scans to a query according to its PDQ priority (among other factors). DS_MAX_SCANS and MAX_PDQPRIORITY allow you to limit the resources that users can assign to a query, according to the following formula:

```
scan_threads = min (nfrags, (DS_MAX_SCANS * (pdqpriority / 100)
                 * (MAX_PDQPRIORITY / 100) )
```

nfrags	is the number of fragments in the table with the largest number of fragments.
pdqpriority	is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement.

For example, suppose a large table contains 100 fragments. With no limit on the number of concurrent scans allowed, Dynamic Server would concurrently execute 100 scan threads to read this table. In addition, as many users as wanted to could initiate this query.

As the Dynamic Server administrator, you set DS_MAX_SCANS to a value lower than the number of fragments in this table to prevent Dynamic Server from being flooded with scan threads by multiple decision-support queries. You can set DS_MAX_SCANS to `20` to ensure that Dynamic Server concurrently executes a maximum of 20 scan threads for parallel scans. Furthermore, if multiple users initiate PDQ queries, each query receives only a percentage of the 20 scan threads, according to the PDQ priority assigned to the query and the MAX_PDQPRIORITY that the Dynamic Server administrator sets.

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 Dynamic Server can run concurrently, count each query that runs with PDQ priority set to `1` or greater as one full query.

Dynamic Server allocates less memory to queries that run with a lower priority, so you can assign lower-priority queries a PDQ priority value that is between `1` and `30`, depending on the resource impact of the query. The total number of queries with PDQ priority values greater than `0` cannot exceed DS_MAX_QUERIES.

Managing Applications

The Dynamic Server administrator, the writer of an application, and the users all have a certain amount of control over the amount of resources that Dynamic Server allocates to processing a query. The Dynamic Server administrator exerts control through the use of configuration parameters. The application developer or the user can exert control through an environment variable or SQL statement.

Using SET EXPLAIN

The output of the SET EXPLAIN statement shows decisions that are made by the query optimizer. It shows whether parallel scans are used, the maximum number of threads required to answer the query, and the type of join used for the query. You can use SET EXPLAIN to study the query plans of an application. You can restructure a query or use OPTCOMPIND to change how the optimizer treats the query.

Using OPTCOMPIND

The **OPTCOMPIND** environment variable and the OPTCOMPIND configuration parameter indicate the preferred join plan, thus assisting the optimizer in selecting the appropriate join method for parallel database queries.

You can influence the optimizer in its choice of a join plan by setting the OPTCOMPIND configuration parameter. The value that you assign to this configuration parameter is referenced only when applications do not set the **OPTCOMPIND** environment variable.

You can set OPTCOMPIND to `0` if you want Dynamic Server to select a join plan exactly as it did in versions of the database server prior to 6.0. This option ensures compatibility with previous versions of the database server.

When you set this parameter, remember that an application with an isolation mode of Repeatable Read can lock all records in a table when it performs a hash join. For this reason, Informix recommends that you set OPTCOMPIND to `1`.

If you want the optimizer to make the determination for you based on costs, regardless of the isolation-level setting of applications, set OPTCOMPIND to `2`.

For more information on OPTCOMPIND and the different join plans, refer to "The Query Plan" on page 7-3.

Using SET PDQPRIORITY

The SET PDQPRIORITY statement allows you to set PDQ priority dynamically within an application. The PDQ priority value can be any integer from `-1` through `100`.

The PDQ priority set with the SET PDQPRIORITY statement supersedes the **PDQPRIORITY** environment variable.

The DEFAULT tag for the SET PDQPRIORITY statement allows an application to revert to the value for PDQ priority as set by the environment variable, if any. For more information about the SET PDQPRIORITY statement, refer to the *Informix Guide to SQL: Syntax*.

User Control of Resources

To indicate the PDQ priority of a query, a user sets the **PDQPRIORITY** environment variable or executes the SET PDQPRIORITY statement prior to issuing a query. In effect, this allows users to request a certain amount of parallel-processing resources for the query.

The resources that a user requests and the amount that Dynamic Server allocates for the query can differ. This difference occurs when the Dynamic Server administrator uses the MAX_PDQPRIORITY configuration parameter to put a ceiling on user-requested resources, as explained in the following section.

Dynamic Server Administrator Control of Resources

To manage the total amount of resources that Dynamic Server allocates to PDQ queries, the Dynamic Server administrator sets the environment variable and configuration parameters that are discussed in the following sections.

Controlling Resources Allocated to PDQ

First, you can set the **PDQPRIORITY** environment variable. The queries that do not set the **PDQPRIORITY** environment variable before they issue a query do not use PDQ. In addition, you can place a ceiling on user-specified PDQ priority levels by setting the MAX_PDQPRIORITY configuration parameter.

When you set the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter, you exert control over the resources that Dynamic Server allocates between OLTP and DSS applications. For example, if OLTP processing is particularly heavy during a certain period of the day, you might want to set MAX_PDQPRIORITY to 0. This configuration parameter puts a ceiling on the resources requested by users who use the **PDQPRIORITY** environment variable, so PDQ is turned off until you reset MAX_PDQPRIORITY to a nonzero value.

Controlling Resources Allocated to Decision-Support Queries

You control the resources that Dynamic Server allocates to decision-support queries by setting the DS_TOTAL_MEMORY, DS_MAX_SCANS, and DS_MAX_QUERIES configuration parameters. In addition to setting limits for decision-support memory and the number of decision-support queries that can run concurrently, Dynamic Server uses these parameters to determine the amount of memory to allocate to individual decision-support queries as they are submitted by users. To do this, Dynamic Server first calculates a unit of memory called a *quantum* by dividing DS_TOTAL_MEMORY by DS_MAX_QUERIES. When a user issues a query, Dynamic Server allocates a percent of the available quanta equal to the PDQ priority of the query.

You can also limit the number of concurrent decision-support scans that Dynamic Server allows by setting the DS_MAX_SCANS configuration parameter.

UNIX

Previous versions of the database server allowed you to set a PDQ priority configuration parameter in the ONCONFIG file. If your applications depend on a global setting for PDQ priority, you can define **PDQPRIORITY** as a shared environment variable in the **informix.rc** file. For more information on the **informix.rc** file, see the *Informix Guide to SQL: Reference*. ♦

Monitoring PDQ Resources

Monitor the resources (shared memory and threads) that the MGM has allocated for PDQ queries, and the resources that those queries currently use.

You monitor PDQ resource use in the following ways:

- Run individual **onstat** utility commands to capture information about specific aspects of a running query.
- Execute a SET EXPLAIN statement before you run a query to write the query plan to an output file.

Using the onstat Utility

You can use various **onstat** utility commands to determine how many threads are active and the shared-memory resources that those threads use.

Monitoring MGM Resources

You can use the **onstat -g mgm** option to monitor how MGM coordinates memory use and scan threads. The **onstat** utility reads shared-memory structures and provides statistics that are accurate at the instant that the command executes. Figure 9-1 on page 9-18 shows sample output.

The **onstat -g mgm** display uses a unit of memory called a *quantum*. The *memory quantum* represents a unit of memory, as follows:

```
memory quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

The *scan thread quantum* is always equal to 1.

```
Memory Grant Manager (MGM)
--------------------------

MAX_PDQPRIORITY:  100
DS_MAX_QUERIES:    5
DS_MAX_SCANS:      10
DS_TOTAL_MEMORY:   4000 KB

Queries:   Active     Ready   Maximum
                3         0         5

Memory:     Total      Free   Quantum
(KB)         4000      3872       800

Scans:      Total      Free   Quantum
               10         8         1

Load Control:    (Memory)      (Scans)  (Priority)  (Max Queries)   (Reinit)
                   Gate 1       Gate 2      Gate 3         Gate 4     Gate 5
(Queue Length)          0            0           0              0          0

Active Queries:
---------------
Session    Query  Priority  Thread   Memory  Scans      Gate
      7  a3d0c0      1     a8adcc    0/0       1/1       -
      7  a56eb0      1     ae6800    0/0       1/1       -
      9  a751d4      0     96b1b8   16/16      0/0       -

Ready Queries:  None

Free Resource        Average #       Minimum #
--------------    ---------------     ---------
Memory             489.2 +- 28.7         400
Scans                8.5 +- 0.5            8

Queries              Average #       Maximum #    Total #
--------------    ---------------     ---------    -------
Active               1.7 +- 0.7            3         23
Ready                0.0 +- 0.0            0          0

Resource/Lock Cycle Prevention count:  0
```

Figure 9-1
onstat -g mgm Output

The first portion of the display shows the values of the PDQ configuration parameters.

The second portion of the display describes MGM internal control information. It includes four groups of information.

The first group is indicated by **Queries.**

Active	Number of PDQ queries that are currently executing
Ready	Number of user queries ready to run, but whose execution Dynamic Server deferred for load-control reasons
Maximum	Maximum number of queries that Dynamic Server permits to be active. Reflects current value of the DS_MAX_QUERIES configuration parameter

The next group is indicated by **Memory**.

Total	Kilobytes of memory available for use by PDQ queries (DS_TOTAL_MEMORY specifies this value.)
Free	Kilobytes of memory for PDQ queries not currently in use
Quantum	Kilobytes of memory in a memory quantum

The next group is indicated by **Scans**.

Total	The total number of scan threads as specified by the DS_MAX_SCANS configuration parameter
Free	Number of scan threads currently available for decision-support queries
Quantum	The number of scan threads in a scan-thread quantum

The last group in this portion of the display describes MGM **Load Control**.

Memory	Number of queries that are waiting for memory
Scans	Number of queries that are waiting for scans
Priority	Number of queries that are waiting for queries with higher PDQ priority to run
Max Queries	Number of queries that are waiting for a query slot
Reinit	Number of queries that are waiting for running queries to complete after an **onmode -M** or **-Q**

The next portion of the display (**Active Queries**) describes the MGM active and ready queues. This portion of the display shows the number of queries waiting at each gate.

Session	The session ID for the session that initiated the query
Query	Address of the internal control block associated with the query
Priority	PDQ priority assigned to the query
Thread	Thread that registered the query with MGM

(1 of 2)

Memory	Memory currently granted to the query / memory reserved for the query (Unit is MGM pages, which is 8 kilobytes.)
Scans	Number of scan threads currently used by the query / number of scan threads allocated to the query
Gate	Gate number at which query is waiting

(2 of 2)

The next portion of the display (**Free Resource**) provides statistics concerning MGM free resources. The numbers in this portion and in the final portion reflect statistics since system initialization or the last **onmode -Q**, **-M**, or **-S**.

Average	Average of memory / scans
Minimum	Available memory / scans

The last portion of the display (**Queries**) provides statistics concerning MGM queries.

Average	Average active / ready queue length
Minimum	Minimum active / ready queue length
Total	Total active / ready queue length

Monitoring PDQ Threads

To obtain information on all of the threads that are running for a decision-support query, use the **onstat -u** and **onstat -g ath** options.

The **onstat -u** option lists all the threads for a session. If a session is running a decision-support query, the primary thread and any additional threads are listed. For example, session `10` in Figure 9-2 on page 9-21 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 9-2
onstat -u Output

The **onstat -g ath** option display also lists these threads and includes a **name** column that indicates the role of the thread. Threads that have been started by a primary decision-support thread have a name that indicates their role in the decision-support query. For example, Figure 9-3 lists four *scan* threads, started by a primary (**sqlexec**) thread.

```
Threads:
tid     tcb     rstcb   prty    status                  vp-class   name
.
.
.
11      994060   0       4      sleeping(Forever)         1cpu    kaio
12      994394   80f2a4  2      sleeping(secs: 51)        1cpu    btclean
26      99b11c   80f630  4      ready                     1cpu    onmode_mon
32      a9a294   812b64  2      ready                     1cpu    sqlexec
113     b72a7c   810b78  2      ready                     1cpu    sqlexec
114     b86c8c   81244c  2      cond wait(netnorm)        1cpu    sqlexec
115     b98a7c   812ef0  2      cond wait(netnorm)        1cpu    sqlexec
116     bb4a24   80fd48  2      cond wait(netnorm)        1cpu    sqlexec
117     bc6a24   81161c  2      cond wait(netnorm)        1cpu    sqlexec
118     bd8a24   811290  2      ready                     1cpu    sqlexec
119     beae88   810f04  2      cond wait(await_MC1)      1cpu    scan_1.0
120     a8ab48   8127d8  2      ready                     1cpu    scan_2.0
121     a96850   810460  2      ready                     1cpu    scan_2.1
122     ab6f30   8119a8  2      running                   1cpu    scan_2.2
```

Figure 9-3
onstat -g ath Output

Monitoring Resources Allocated for a Sessions

Use the **onstat -g ses** option to monitor the resources allocated for, and used by, a session that is running a decision-support query. The **onstat -g ses** option 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 decision-support query
- The number of threads that the database server started for a session

For example, in Figure 9-4, session number 49 is running five threads for a decision-support query.

Figure 9-4
onstat -g ses Output

```
session                                      #RSAM    total       used
id       user     tty      pid   hostname threads  memory      memory
57       informix -        0     -        0        8192        5908
56       user_3   ttyp3    2318  host_10  1        65536       62404
55       user_3   ttyp3    2316  host_10  1        65536       62416
54       user_3   ttyp3    2320  host_10  1        65536       62416
53       user_3   ttyp3    2317  host_10  1        65536       62416
52       user_3   ttyp3    2319  host_10  1        65536       62416
51       user_3   ttyp3    2321  host_10  1        65536       62416
49       user_1   ttyp2    2308  host_10  5        188416      178936
2        informix -        0     -        0        8192        6780
1        informix -        0     -        0        8192        4796
```

Using SET EXPLAIN

When PDQ is turned on, the SET EXPLAIN output shows whether the optimizer chose parallel scans. If the optimizer chose parallel scans, the output shows PARALLEL. If PDQ is turned off, the output shows SERIAL.

If PDQ is turned on, the optimizer indicates the maximum number of threads that are required to answer the query. The output shows # of Secondary Threads. This field indicates the number of threads that are required in addition to your user session thread. The total number of threads necessary is the number of secondary threads plus 1.

The following example shows the SET EXPLAIN output for a table with fragmentation and PDQ priority set to LOW:

```
select * from t1 where c1 > 20

Estimated Cost: 2
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Parallel, fragments: 2)

 Filters: informix.t1.c1 > 20

# of Secondary Threads = 1
```

The following example of SET EXPLAIN output shows a query with a hash join between two fragmented tables and PDQ priority set to ON. The query is marked with DYNAMIC HASH JOIN, and the table on which the hash is built is marked with `Build Outer`.

```
QUERY:
------
select h1.c1, h2.c1 from h1, h2 where h1.c1 = h2.c1

Estimated Cost: 2
Estimated # of Rows Returned: 5

1) informix.h1: SEQUENTIAL SCAN (Parallel, fragments: ALL)

2) informix.h2: SEQUENTIAL SCAN (Parallel, fragments: ALL)


DYNAMIC HASH JOIN (Build Outer)
 Dynamic Hash Filters: informix.h1.c1 = informix.h2.c1

# of Secondary Threads = 6
```

The following example of SET EXPLAIN output shows a table with fragmentation, with PDQ priority set to LOW, and an index that was selected as the access plan:

```
select * from t1 where c1 < 13

Estimated Cost: 2
Estimated # of Rows Returned: 1

1) informix.t1: INDEX PATH

 (1) Index Keys: c1 (Parallel, fragments: ALL)
 Upper Index Filter: informix.t1.c1 < 13


# of Secondary Threads = 3
```

Chapter

10 Improving Individual Query Performance

This chapter suggests ways to apply the general and conceptual information in addition to the monitoring information that is provided in this manual.

Topics discussed include:

- Eliminating table fragments (see "Query Expressions for Fragment Elimination" on page 6-22)
- Improving the selectivity of filters
- Creating data distributions to improve optimizer performance
- Improving index performance
- Improving sequential scans
- Reducing the impact of join and sort operations
- Reviewing the optimization level

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. Then examine your data model and see if changes that are suggested in this chapter will improve it.

"How to Display the Query Plan" on page 7-14 provides an example of using SET EXPLAIN.

Using a Dedicated Test System

If possible, you might decide to test a query 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 can 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, sorting 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 conclusion 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 with the following goals in mind:

- 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. Initiate tests when the system load is consistently light so that you are truly measuring the impact of your query only.
- If the query is embedded in a complicated program, you can extract the SELECT statement and embed it in a DB-Access script.

Improving Filter Selectivity

The greater the precision with which you specify the desired rows, the greater the likelihood that your queries will complete quickly. You control the amount of information that the query evaluates using the WHERE clause of the SELECT statement. The conditional expression given in the WHERE clause is commonly called a *filter*.

"Assessing Filters" on page 7-20 describes how filter selectivity affects the query plan that the optimizer chooses.

Avoid the following types of filters for best performance:

- Certain difficult regular expressions
- Noninitial substrings

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 Dynamic 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 Dynamic Server to examine every value in the column:

```
SELECT * FROM customer WHERE fname NOT LIKE '%y'
```

An index cannot be used with such a filter, so the table in this example must be accessed 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 desired 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 customer
    WHERE zipcode[4,5] > '50'
```

An index cannot be used to evaluate such a filter.

The optimizer uses an index to process a filter that tests an initial substring of an indexed column. However, the presence of the substring test can interfere with the use of a composite index to test both the substring column and another column.

Updating Statistics

The UPDATE STATISTICS statement updates the statistics in the system catalogs that the optimizer uses to determine the lowest-cost query plan. To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals.

Run UPDATE STATISTICS LOW as often as necessary to ensure that the statistic for the number of rows is as up-to-date as possible. Therefore, if the cardinality of a table changes often, run the statement more often for that table. Run UPDATE STATISTICS LOW, which is the default mode, on the table.

```
UPDATE STATISTICS FOR TABLE tab1;
```

Creating Data Distributions

You can use the MEDIUM or HIGH keywords with the UPDATE STATISTICS statement to specify the mode for data distributions on specific columns. These keywords indicate that Dynamic Server is to generate statistics about the distribution of data values for each specified column and place that information in a system catalog table called **sysdistrib**. 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. Data distributions in **sysdistrib** supercede values in the **colmin** and **colmax** column of the **syscolumns** system catalog table when the optimizer estimates the number of rows returned.

When you use data-distribution statistics for the first time, try to update statistics in MEDIUM mode for all your tables, and then update statistics in HIGH mode for all columns that head indexes. This strategy produces statistical query estimates for the columns that you specify. These estimates, on the average, have a 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 the MEDIUM mode. The default percent value for MEDIUM mode is 2.5 percent. (For columns with HIGH mode distributions, the default resolution is 0.1 percent.)

Unless column values change considerably, you do not need to regenerate the data distributions. You can verify the accuracy of the distribution by comparing **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
```

For each table that your query accesses, build data distributions according to the following guidelines:

1. Run UPDATE STATISTICS MEDIUM for all columns in a table that *do not* head an index. This step is a single UPDATE STATISTICS statement. The default parameters are sufficient unless the table is very large, in which case you should 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.

2. Run UPDATE STATISTICS HIGH for all columns that head an index. For the fastest execution time of the UPDATE STATISTICS statement, you *must* 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.

3. 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.

4. For small tables, run 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 the *Informix Guide to SQL: Syntax*.

Updating Statistics for Join Columns

Because of improvements and adjusted cost estimates to establish better query plans, the optimizer depends greatly on an accurate understanding of the underlying data distributions in certain cases. You might still feel that a complex query does not execute quickly enough, even though you followed the guidelines in "Creating Data Distributions" on page 10-7. If your query involves equality predicates, 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. If you followed the guidelines in "Creating Data Distributions" on page 10-7, 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.
  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 your table is very large, UPDATE STATISTICS with the HIGH mode can take a long time to execute.*

The following example shows a query that involves join columns:

```
SELECT employee.name, address.city
    FROM employee, address
    WHERE employee.ssn = address.ssn
    AND employee.name = 'James'
```

In this example, the 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 plan and execution order.

You cannot use the UPDATE STATISTICS statement to create data distributions for a table that is external to the current database. For additional information about data distributions and the UPDATE STATISTICS statement, see the *Informix Guide to SQL: Syntax*.

UPDATE STATISTICS Performance Considerations

When you execute the UPDATE STATISTICS statement, the database server uses memory and disk to sort and construct data distributions. You can affect the amount of memory and disk available for UPDATE STATISTICS with the following methods:

- PDQ priority

  Although the UPDATE STATISTICS statement is not processed in parallel, you can obtain more memory for sorting when you set PDQ priority greater than 0. The default value for PDQ priority is 0. You set PDQ priority with either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

  For more information on PDQ priority, refer to "Allocating Resources for PDQ Queries" on page 9-7.

- **DBUPSPACE** environment variable

  You can use the **DBUPSPACE** environment variable to constrain the amount of system disk space that the UPDATE STATISTICS statement can use to construct multiple column distributions simultaneously.

  For more information on this environment variable, refer to the *Informix Guide to SQL: Reference*.

How to Improve Performance with Indexes

You can often improve the performance of a query by adding, or in some cases removing, indexes. Consider using some of the methods that are described in the following sections to improve the performance of a query.

Replacing Autoindexes with Permanent Indexes

If the query plan includes an *autoindex* path to a large table, take it as a recommendation from the optimizer that you can improve performance by adding an index on that column. It is reasonable to let Dynamic Server build and discard an index if you perform the query occasionally, but if you perform a query regularly, you can save time by creating a permanent index.

Using Composite Indexes

The optimizer can use a composite index (one that covers more than one column) in several ways. An index on the columns *a*, *b*, and *c* (in that order) can be used in the following ways:

- To locate a particular row

  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 that composite index:

  ```
  where b=10
  where c=221
  where a>=12 and b=15
  ```

- To replace a table scan when all of the desired columns are contained within the index

  A scan that uses the index but does not reference the table is termed a *key-only search*.

- To join column *a*, columns *ab*, or columns *abc* to another table
- To implement ORDER BY or GROUP BY on columns *a*, *ab*, or *abc*, but not on *b*, *c*, *ac*, or *bc*

Execution is most efficient when you create a composite index with the columns in order from most to least distinct. In other words, the column that returns the highest count of distinct rows when queried using the DISTINCT keyword of the SELECT statement should come first in the composite index.

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, you must create a composite index on (**a**, **b** DESC) or on (**a** DESC, **b**). You need to create only one of these indexes because of the bidirectional traversal capability of the database server. For more information on the bidirectional traversal capability, refer to the *Informix Guide to SQL: Syntax*.

On the other hand, it can be less expensive to do a table scan and sort the results instead of using the composite index when the following criteria are met:

- Your table is well ordered relative to your index.
- The number of rows retrieved by the query represents a large percentage of the available data.

Using Indexes for Data Warehousing Applications

Many data warehouse databases use a *star schema*. A star schema consists of a *fact* table and a number of *dimensional* tables. The fact table is generally very large and contains the quantitative or factual information about the subject. A dimensional table describes an attribute in the fact table.

When a dimension needs lower level information, the dimension is modeled by a hierarchy of tables, called a *snowflake schema*.

For more information on star schemas and snowflake schemas, refer to the *Informix Guide to Database Design and Implementation*.

Queries that use tables in a star schema or snowflake schema can benefit from the proper index on the fact table.

Consider the example of a star schema with one fact table named **orders** and four dimensional tables named **customers**, **suppliers**, **products**, and **clerks**. The **orders** table describes the details of each sale order, which includes the customer ID, supplier ID, product ID, and sales clerk ID. Each dimensional table describes an ID in detail. The **orders** table is very large and the four dimensional tables are small.

The following query finds the total direct sales revenue in the Menlo Park region (postal code 94025) for hard drives supplied by the Johnson supplier:

```
SELECT sum(orders.price)
FROM orders, customers, suppliers,product,clerks
WHERE orders.custid = customers.custid
    AND customers.zipcode = 94025
    AND orders.suppid = suppliers.suppid
    AND suppliers.name = 'Johnson'
    AND orders.prodid = product.prodid
    AND product.type = 'hard drive'
    AND orders.clerkid = clerks.clerkid
    AND clerks.dept = 'Direct Sales'
```

This query uses a typical star join, in which the fact table joins with all dimensional tables on a foreign key. Each dimensional table has a very selective table filter.

An optimal plan for the star join is to perform a cartesian product on the four dimensional tables and then join the result with the fact table. The following index on the fact table allows the optimizer to choose the optimal query plan:

```
CREATE INDEX ON orders(custid,suppid,prodid,clerkid)
```

Without this index, the optimizer might choose to first join the fact table with one dimensional table and then join the result with the remaining dimensional tables. The optimal plan provides better performance.

Dropping and Rebuilding Indexes After Updates

When an update transaction commits, the Dynamic Server btree cleaner removes deleted index entries and, if necessary, rebalances the index nodes. However, depending on your application (in particular, the order in which it adds and deletes keys from the index), the structure of an index can become inefficient.

Use the **oncheck -pT** command 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 want to drop and recreate the index with a larger value for FILLFACTOR to make the unused disk space available.

For more information on how Dynamic Server maintains an index tree, refer to the information on disk structure and storage in your *Administrator's Guide*.

Improving Sequential Scans

You can improve performance of sequential read operations on large tables by eliminating repeated sequential scans.

Sequential access to a table other than the very first table in the plan is ominous because it threatens to read every row of the table once for every row selected from the preceding tables. You should be able to judge how many times that is: perhaps a few, but perhaps hundreds or even thousands.

If the table is small, it is harmless to read it repeatedly because the table resides completely in memory. 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 pushes other useful pages out of the buffers.

When the table is larger than a few pages, however, repeated sequential access is deadly to performance. One way to prevent this problem is to provide an index to the column that is used to join the table.

Any user with the Resource privilege can build additional indexes. Use the CREATE INDEX statement to make an index.

An index consumes disk space proportional to the width of the key values and the number of rows. (See "Estimating Table and Index Size" on page 4-8.) Also, Dynamic Server must update the index whenever rows are inserted, deleted, or updated; this step slows these operations. If necessary, you can use the DROP INDEX statement to release the index after a series of queries, which frees space and makes table updates easier.

Reducing the Impact of Join and Sort Operations

After you understand what the query is doing, 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 parallel sorts.
- Use temporary tables to reduce sorting scope.

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 infrequent sorts or sorts of relatively small numbers of output rows.

Try to 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 by using an index. The following factors prevent the optimizer from using 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 the ORDER BY or GROUP BY clause.
- The ordered columns are taken from different tables.

Another way to avoid sorts is discussed in a following section, "Using Temporary Tables to Reduce Sorting Scope."

If a sort is necessary, look for ways to simplify it. As discussed in "Sort-Time Costs" on page 7-24, the sort is quicker if you can sort on fewer or narrower columns.

Using Parallel Sorts

When you cannot avoid sorting, Dynamic Server takes advantage of multiple CPU resources to perform the required sort-and-merge operations in parallel. Dynamic Server can use parallel sorts for any query; parallel sorts are not limited to PDQ queries. The **PSORT_NPROCS** environment variable specifies the maximum number of threads that can be used to sort a query.

When PDQ priority is greater than 0 and **PSORT_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable to request a specific proportion of PDQ resources for a query. You can use the MAX_PDQPRIORITY parameter to limit the number of such user requests. For more information on MAX_PDQPRIORITY, refer to "MAX_PDQPRIORITY" on page 3-15.

In some cases, the amount of data being sorted can overflow the memory resources allocated to the query, resulting in I/O to a dbspace or sort file. For more information, refer to "Dbspaces for Temporary Tables and Sort Files" on page 3-50.

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 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 occurs, each of the following form (using hypothetical table and column names):

```
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 specified postal code, Dynamic Server searches the index on **rcvbles.customer_id** and performs a nonsequential disk access for every match. The rows are written to a temporary file and sorted. For more information on temporary files, refer to "Dbspaces for Temporary Tables and Sort Files" on page 3-50.

This procedure is acceptable if the query is performed only once, but this example includes a series of queries, each incurring the same amount of work.

An alternative is to select all customers with outstanding balances into a temporary table, ordered by customer name, as the following example shows:

```
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
```

Now 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.

Reviewing the Optimization Level

You normally obtain optimum overall performance with the default optimization level, high. The time that it takes to optimize the statement is usually unimportant. However, if experimentation with your application reveals that your query is still taking too long, you can set your optimization level to low and then check the SET EXPLAIN output to see if the optimizer chose the same query plan as before.

You can specify a high or low level of database server optimization with the SET OPTIMIZATION statement. This statement is described in detail in the *Informix Guide to SQL: Syntax*.

Optimizing User-Response Time for Queries

The two types of optimization goals for query performance are as follows:

- Optimizing total query time
- Optimizing user-response time

Total query time is the time it takes to return all rows to the application. Total query time is most important for batch processing or for queries that require all rows be processed before returning a result to the user, as in the following query:

```
SELECT count(*) FROM orders
WHERE order_amount > 2000;
```

User-response time is the time that it takes for the database server to return a screen-full of rows back to an interactive application. In interactive applications, only a screenful of data can be requested at one time. For example, the user application can only display 10 rows at one time for the following query:

```
SELECT * FROM orders
WHERE order_amount > 2000;
```

Which optimization goal is more important can have an effect on the query path that the optimizer chooses. For example, the optimizer might choose a nested-loop join instead of a hash join to execute a query if user-response time is most important, even though a hash join might result in a reduction in total query time.

How to Specify the Query Performance Goal

The default behavior is for the optimizer to choose query plans that optimize the total query time. You can specify user-response-time optimization at several different levels:

- For the Dynamic Server system

  Set the OPT_GOAL configuration parameter to `0` for user-response-time optimization, as in the following example:

  ```
  OPT_GOAL 0
  ```

  Set OPT_GOAL to `-1` for total-query-time optimization.

- For the user environment

  The **OPT_GOAL** environment variable can be set before the user application starts.

  UNIX

  Set the **OPT_GOAL** environment variable to `0` for user-response-time optimization, as in the following sample commands:

  ```
  Bourne shell        OPT_GOAL = 0
                      export OPT_GOAL

  C shell             setenv OPT_GOAL 0   ♦
  ```

  Set the **OPT_GOAL** environment variable to `-1` for total-query-time optimization.

- Within the session

  You can control the optimization goal with the SET OPTIMIZATION statement in SQL. The optimization goal set with this statement stays in effect until the session ends or until another SET OPTIMIZATION statement changes the goal.

  The following statement causes the optimizer to choose query plans that favor total-query-time optimization:

  ```
  SET OPTIMIZATION ALL_ROWS
  ```

  The following statement causes the optimizer to choose query plans that favor user-response-time optimization:

  ```
  SET OPTIMIZATION FIRST_ROWS
  ```

- For individual queries

  You can use FIRST_ROWS and ALL_ROWS optimizer directives to instruct the optimizer which query goal to use. For more information about these directives, refer to "Optimization Goal Directives" on page 8-10.

The precedence for these levels is as follows:

- Optimizer directives
- SET OPTIMIZATION statement
- **OPT_GOAL** environment variable
- OPT_GOAL configuration parameter

For example, optimizer directives take precedence over the goal specified by the SET OPTIMIZATION statement.

Preferred Query Plans for User-Response-Time Optimization

When the optimizer chooses query plans to optimize user-response time, it computes the cost to retrieve the first row in the query for each plan and chooses the plan with the lowest cost. In some cases, the query plan that costs the least to retrieve the first row might not be the optimal path to retrieve all rows in the query.

Some of the possible differences in query plans are explained in the following sections.

Nested-Loop Joins Versus Hash Join

Hash joins generally have a higher cost to retrieve the first row than nested-loop joins do. The database server must build the hash table before it retrieves any rows. However, in some cases, total query time is faster if the database server uses a hash join.

In the following example, **tab2** has an index on **col1**, but **tab1** does not have an index on **col1**. When you execute SET OPTIMIZATION ALL_ROWS before you run the query, the database server uses a hash join and ignores the existing index as the following SET EXPLAIN output shows:

```
QUERY:
------
select * from tab1,tab2
where tab1.col1 = tab2.col1
Estimated Cost: 125
Estimated # of Rows Returned: 510
1) lsuto.tab2: SEQUENTIAL SCAN
2) lsuto.tab1: SEQUENTIAL SCAN
DYNAMIC HASH JOIN
    Dynamic Hash Filters: lsuto.tab2.col1 = lsuto.tab1.col1
```

However, when you execute SET OPTIMIZATION FIRST_ROWS before you run the query, the database server uses a nested-loop join. The clause `(FIRST_ROWS OPTIMIZATION)` in the following SET EXPLAIN output shows that the optimizer used user-response-time optimization for the query:

```
QUERY:        (FIRST_ROWS OPTIMIZATION)
------
select * from tab1,tab2
where tab1.col1 = tab2.col1
Estimated Cost: 145
Estimated # of Rows Returned: 510
1) lsuto.tab1: SEQUENTIAL SCAN
2) lsuto.tab2: INDEX PATH
    (1) Index Keys: col1
        Lower Index Filter: lsuto.tab2.col1 = lsuto.tab1.col1
NESTED LOOP JOIN
```

Table Scans Versus Index Scans

In cases where the database server returns a large number of rows from a table, the lower cost option for the total-query-time goal might be to scan the table instead of using an index. However, to retrieve the first row, the lower cost option for the user response time goal might be to use the index to access the table.

Ordering with Fragmented Indexes

When an index is not fragmented, the database server can use the index to avoid a sort. For more information on avoiding sorts, refer to "Avoiding or Simplifying Sort Operations" on page 10-15. However, when an index is fragmented, the ordering can only be guaranteed within the fragment, not between fragments.

Usually, the least expensive option for the total-query-time goal is to scan the fragments in parallel and then use the parallel sort to produce the proper ordering. However, this option does not favor the user-response-time goal.

Instead, if the user-response time is more important, the database server reads the index fragments in parallel and merges the data from all of the fragments. No additional sort is generally needed.

Chapter

The onperf Utility on UNIX

11

This chapter describes the **onperf** utility, a windowing environment that you can use to monitor Dynamic Server performance. The **onperf** utility monitors Dynamic Server running on the UNIX operating system.

The **onperf** utility allows you to monitor most of the same Dynamic Server metrics that the **onstat** utility reports. The **onperf** utility provides these main advantages over **onstat**:

- Displays metric values graphically in real time
- Allows you to choose which metrics to monitor
- Allows you to scroll back to previous metric values to analyze a trend
- Can save performance data to a file for review at a later time

Overview of the onperf Utility

This section provides an overview of **onperf** functionality and the different **onperf** tools.

Basic onperf Functions

The **onperf** utility performs the following basic functions:

- Displays the values of Dynamic Server metrics in a tool window
- Saves Dynamic Server metric values to a file
- Allows review of Dynamic Server metric values from a file

Displaying Metric Values

When **onperf** starts, it activates the following processes:

- **The onperf process.** This process controls the display of **onperf** tools.
- **The data-collector process.** This process attaches to shared memory and passes performance information to the **onperf** process for display in an **onperf** tool.

An **onperf** tool is a Motif window that is managed by an **onperf** process, as Figure 11-1 shows.

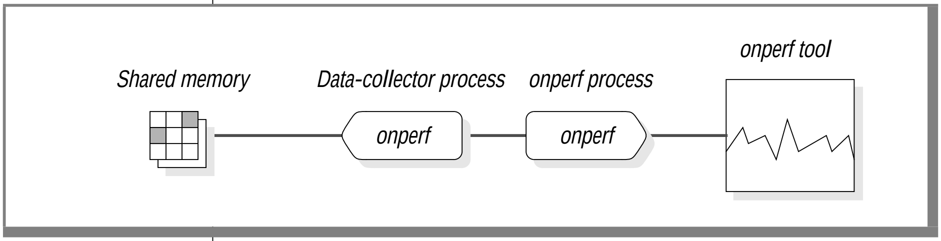


Figure 11-1
Data Flow from Shared Memory to an onperf Tool Window

Saving Metric Values to a File

The **onperf** utility allows designated metrics to be continually buffered. The data collector writes these metrics to a circular buffer called the *data-collector buffer*. When the buffer becomes full, the oldest values are overwritten as the data collector continues to add data. The current contents of the data-collector buffer are saved to a history file, as Figure 11-2 illustrates.

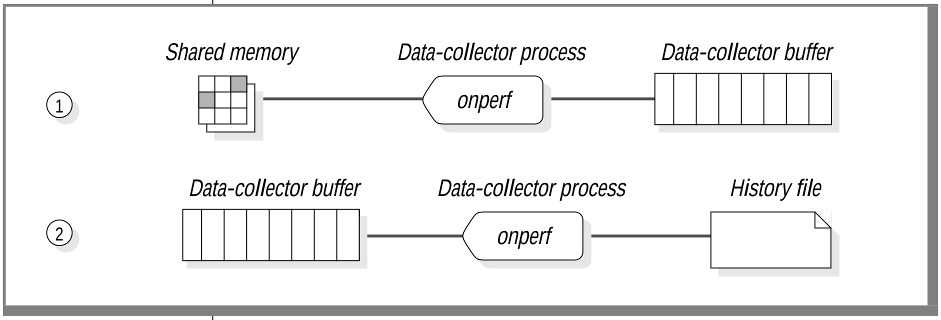


Figure 11-2
How onperf Saves Performance Data

The **onperf** utility uses either a binary format or an ASCII representation for data in the history file. The binary format is host dependent and allows data to be written quickly. The ASCII format is portable across platforms.

You have control over the set of metrics stored in the data-collector buffer and the number of samples. You could buffer all metrics; however, this action might consume more memory than is feasible. A single metric measurement requires 8 bytes of memory. For example, if the sampling frequency is one sample per second, then to buffer 200 metrics for 3,600 samples requires approximately 5.5 megabytes of memory. If this process represents too much memory, you must reduce the depth of the data-collector buffer, the sampling frequency, or the number of buffered metrics.

You can use the Configuration dialog box to configure the buffer depth or the sampling frequency. For more information on the Configuration dialog box, refer to "Graph Tool Configure Menu and the Configuration Dialog Box" on page 11-14.

Reviewing Metric Measurements

You can review the contents of a history file in a tool window. When you request a tool to display a history file, the **onperf** process starts a playback process that reads the data from disk and sends the data to the tool for display. The playback process is similar to the data-collector process mentioned under "Saving Metric Values to a File" on page 11-4. However, instead of reading data from shared memory, the playback process reads measurements from a history file. The playback process is shown in Figure 11-3.

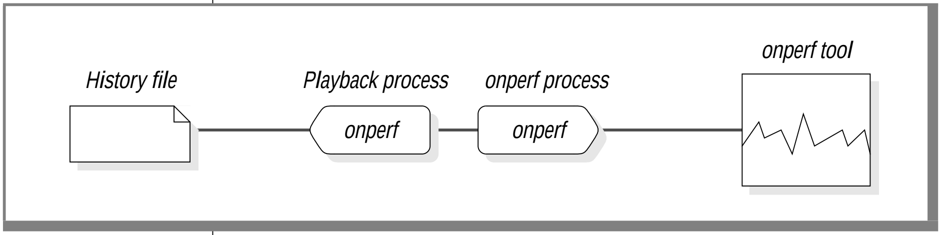


Figure 11-3
Flow of Data from a History File to an onperf Tool Window

The onperf Tools

The **onperf** utility provides the following Motif windows, called *tools*, to display metric values:

- Graph tool

  This tool allows you to monitor general performance activity. You can use this tool to display any combination of metrics that **onperf** supports and to display the contents of a history file. For more information, see "Graph Tool" on page 11-9.

- Query-tree tool

  This tool displays the progress of individual queries. For more information, see "Query-Tree Tool" on page 11-19.

- Status tool

  This tool displays status information about Dynamic Server and allows you to save the data that is currently held in the data-collector buffer to a file. For more information, see "Status Tool" on page 11-20.

- Activity tools

  These tools display specific Dynamic Server activities. Activity tools include disk, session, disk-capacity, physical-processor, and virtual-processor tools. The physical-processor and virtual-processor tools, respectively, display information about all CPUs and VPs. The other activity tools each display the top 10 instances of a resource ranked by a suitable activity measurement. For more information, see "Activity Tools" on page 11-21.

Requirements for Running onperf

When you install Dynamic Server, the following executable files are written to the **$INFORMIXDIR/bin** directory:

- **onperf**
- **onedcu**
- **onedpu**
- **xtree**

In addition, the on-line help file, **onperf.hlp**, is placed in the **$INFORMIXDIR/hhelp** directory.

When Dynamic Server is installed and running in on-line mode, you can bring up **onperf** tools either on the computer that is running Dynamic Server or on a remote computer or terminal that can communicate with your Dynamic Server instance. Both possibilities are illustrated in Figure 11-4. In either case, the computer that is running the **onperf** tools must support the X terminal and the **mwm** window manager.

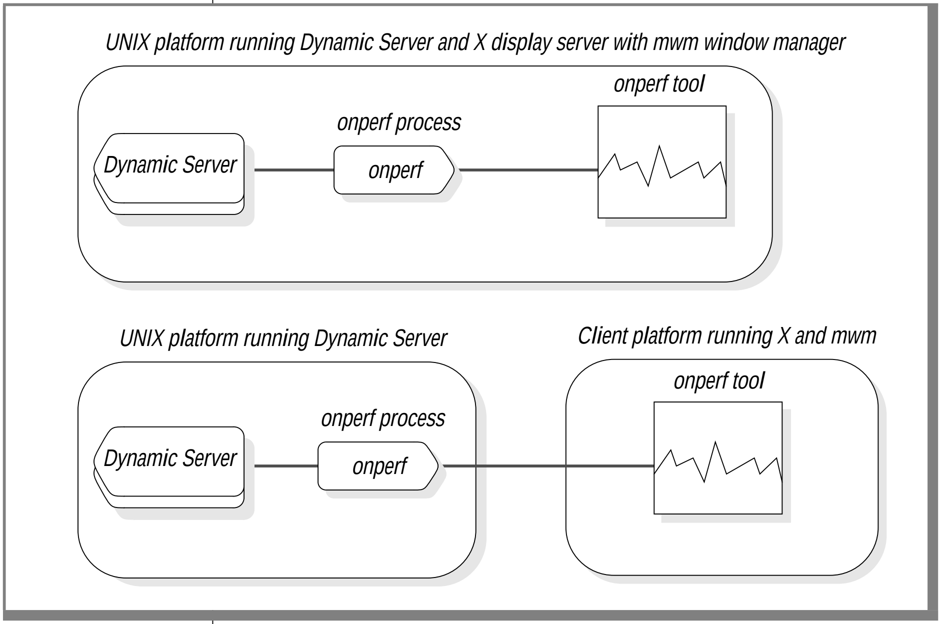

UNIX platform running Dynamic Server and X display server with mwm window manager
onperf tool
onperf process
Dynamic Server
onperf
UNIX platform running Dynamic Server
Client platform running X and mwm
onperf tool
onperf process
Dynamic Server
onperf

Figure 11-4
Two Options for Running onperf

Starting and Exiting onperf

Before you start **onperf**, set the following environment variables to the appropriate values:

- **DISPLAY**
- **LD_LIBRARY_PATH**

Set the **DISPLAY** environment variable as follows:

C shell	`setenv DISPLAY` *displayname*`0:0 #`
Bourne shell	`DISPLAY=`*displayname*`0:0 #`

In these commands, *displayname* is the name of the computer or X terminal where the **onperf** window should appear.

Set the **LD_LIBRARY_PATH** environment variable to the appropriate value for the Motif libraries on the computer that is running **onperf**.

With the environment properly set up, you can enter **onperf** to bring up a graph tool window, as described in "The onperf User Interface" on page 11-9.

To exit **onperf**, use the **Close** option to close each tool window, use the **Exit** option of a tool, or choose **Window Manager→Close**.

You can monitor multiple Dynamic Server database servers from the same Motif client by invoking **onperf** for each database server, as the following example shows:

```
INFORMIXSERVER=instance1 ; export INFORMIXSERVER; onperf
INFORMIXSERVER=instance2 ; export INFORMIXSERVER; onperf
.
.
.
```

The onperf User Interface

When you invoke **onperf**, the utility displays an initial graph-tool window. From this graph-tool window, you can display additional graph-tool windows as well as the query-tree, data-collector, and activity tools. The graph-tool windows have no hierarchy; you can create and close these windows in any order.

Graph Tool

The graph tool is the principal **onperf** interface. This tool allows you to display any set of Dynamic Server metrics that the **onperf** data collector obtains from shared memory. A diagram of a graph tool is shown in Figure 11-5.

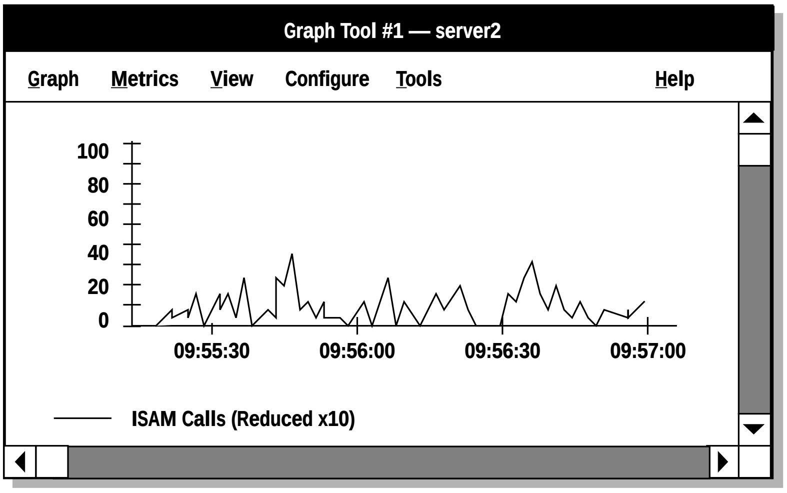


Figure 11-5
Graph Tool Window

You cannot bring up a graph tool window from a query-tree tool, a status tool, or one of the activity tools.

Title Bar

All graph tool windows contain information in the title bar. Figure 11-6 shows the format.

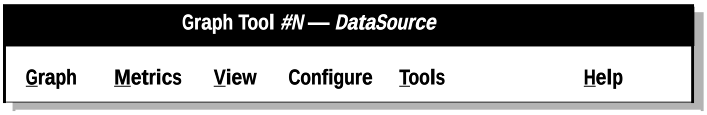


Figure 11-6
Title-Bar Information

When you invoke **onperf**, the initial graph tool window displays a title bar such as the one shown in Figure 11-7. In this case, *serverName* is the database server named by the **INFORMIXSERVER** environment variable.

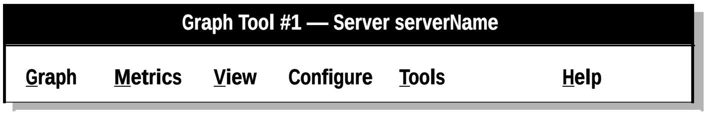


Figure 11-7
Title Bar for Initial Graph Tool Window

Because the configuration of this initial graph tool has not yet been saved or loaded from disk, **onperf** does not display the name of a configuration file in the title bar.

The data source displayed in Figure 11-7 is the database server that the **INFORMIXSERVER** environment variable specifies, meaning that the data comes from the shared memory of the indicated Dynamic Server instance.

Suppose you open a historical data file named **caselog.23April.2PM** in this graph tool window. The title bar now displays the information shown in Figure 11-8.

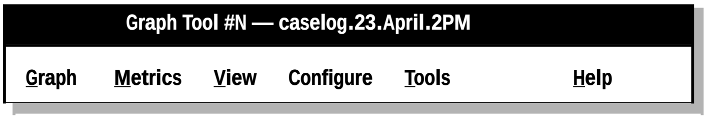


Figure 11-8
Title Bar for a Graph Tool Window That Displays Data from a History File

Graph Tool Graph Menu

The **Graph** menu provides the following options.

Option	Use
New	This option creates a new graph tool. All graph tools that you create using this option share the same data-collector and **onperf** processes. Create new graph tools using this option rather than by invoking **onperf** multiple times.
Open History File	This option loads a previously saved file of historical data into the graph tool for viewing. When you select a file, **onperf** starts a playback process to view the file.
Save History File	This option saves the contents of the data-collector buffer to either an ASCII or a binary file, as specified in the Configuration dialog box.
Save History File As	This option specifies the file name in which to save the contents of the data-collector buffer.
Annotate	This option brings up a dialog box in which you can enter a header label and a footer label. Each label is optional. The labels are displayed on the graph. When you save the graph configuration, **onperf** includes these labels in the saved configuration file.
Print	This option brings up a dialog box that allows you to select a destination file. You cannot send the contents of the graph tool directly to a printer; you must use this option to specify a file and subsequently send the PostScript file to a printer.
Close	This option closes the tool. When a tool is the last remaining tool of the **onperf** session, this menu item behaves like the **Exit** option.
Exit	This option exits **onperf**.

Important: *To save your current configuration before you load a new configuration from a file, you must choose* ***Configure→Save Configuration*** *or* ***Configure→Save Configuration As****.*

Graph Tool Metrics Menu

Use the **Metrics** menu to choose the class of metrics to display in the graph tool.

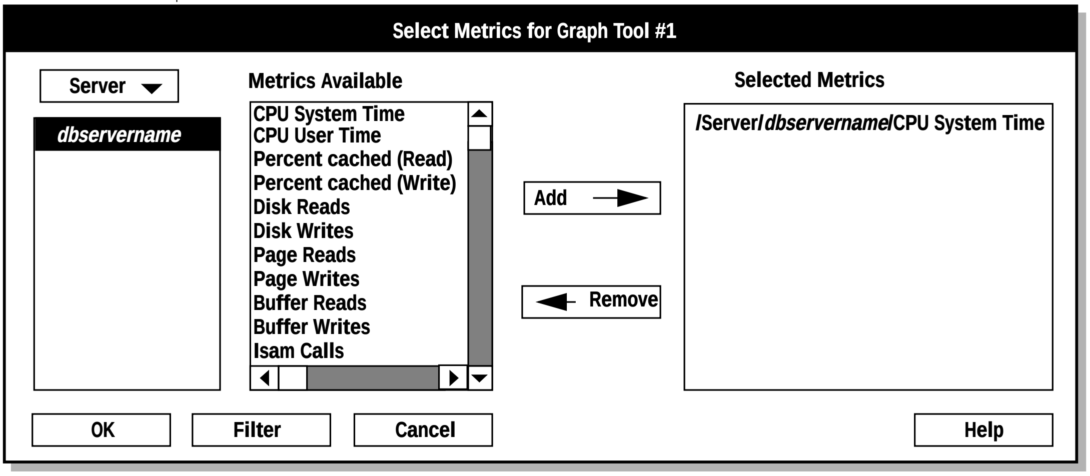
Metrics are organized by *class* and *scope*. When you select a metric for the graph tool to display, you must specify the metric class, the metric scope, and the name of the metric.

The *metric class* is the generic Dynamic Server component or activity that the metric monitors. The *metric scope* depends on the metric class. In some cases, the metric scope indicates a particular component or activity. In other cases, the scope indicates all activities of a given type across an instance of Dynamic Server.

The **Metrics** menu has a separate option for each class of metrics. For more information on metrics, see "Ways to Use onperf" on page 11-22.

When you choose a class, such as **Server**, you see a dialog box like the one shown in Figure 11-9.

Figure 11-9
The Select Metrics Dialog Box



The Select Metrics dialog box contains three list boxes. The list box on the left displays the valid scope levels for the selected metrics class. For example, when the scope is set to `Server`, the list box displays the dbservername of the Dynamic Server database server that is being monitored. When you select a scope from this list, **onperf** displays the individual metrics that are available within that scope in the middle list box. You can select one or more individual metrics from this list and add them to the display.

Tip: *You can display metrics from more than one class in a single graph-tool window. For example, you might first select ISAM Calls, Opens, and Starts from the Server class. When you choose the **Option** menu in the same dialog box, you can select another metric class without exiting the dialog box. For example, you might select the Chunks metric class and add the Operations, Reads, and Writes metrics to the display.*

The **Filter** button in the dialog box brings up an additional dialog box in which you can filter long text strings shown in the Metrics dialog box. The Filter dialog box also lets you select tables or fragments for which metrics are not currently displayed.

Graph Tool View Menu

The **View** menu provides the following options.

Option	Use
Line	This option changes the graph tool to the line format. Line format includes horizontal and vertical scroll bars. The vertical scroll bar adjusts the scale of the horizontal time axis. When you raise this bar, **onperf** reduces the scale and vice versa. The horizontal scroll bar allows you to adjust your view along the horizontal time axis. You can change the color and width of the lines in the line format by clicking on the legend in the graph tool. When you do, **onperf** generates a Customize Metric dialog box that gives you a choice of line color and width.
Horizontal Bar Graph	This option changes the graph tool to the horizontal bar format.
Vertical Bar Graph	This option changes the graph tool to the vertical bar format.

(1 of 2)

Option	Use
Pie	This option changes the graph tool to the pie-chart format. You must select at least two metrics before you can display a pie chart.
Quick Rescale Axis	This option rescales the axis to the largest point that is currently visible on the graph. This button turns off automatic rescaling.
Configure Axis	This option displays the Axis Configuration dialog box. Use this dialog box to select a fixed value for the y-axis on the graph or select automatic axis scaling.

(2 of 2)

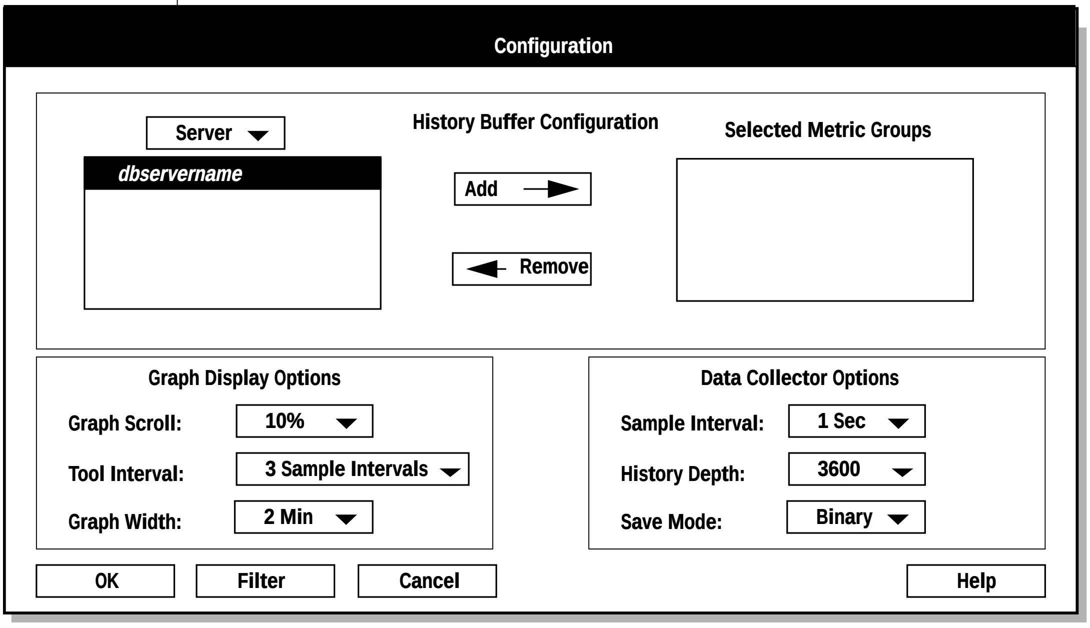
Graph Tool Configure Menu and the Configuration Dialog Box

The **Configure** menu provides the following options.

Option	Use
Edit Configuration	This option brings up the Configuration dialog box, which allows you to change the settings for the current data-collector buffer, graph-tool display options, and data-collector options. The Configuration dialog box is shown in Figure 11-9 on page 11-12.
Open Configuration	This option reinitializes **onperf** using the settings that are stored in the configuration file. Unsaved data in the data-collector buffer is lost.
Save Configuration	This option saves the current configuration to a file. If no configuration file is currently specified, **onperf** prompts for one.
Save Configuration As	This option saves a configuration file; it always prompts for a file name.

You can use the Configuration dialog box (brought up by the **Edit Configuration** option) to configure data-collector options, graph-display options, and metrics about which to collect data.

Figure 11-10
The Configuration Dialog Box



Graph Scroll: 10%
Tool Interval: 3 Sample Intervals
Graph Width: 2 Min
Sample Interval: 1 Sec
History Depth: 3600
Save Mode: Binary

The Configuration dialog box provides the following areas for configuring display options.

Option	Use
History Buffer Configuration	This area allows you to select a metric class and metric scope to include in the data-collector buffer. The data collector gathers information about all metrics that belong to the indicated class and scope.
Graph Display Options	The options listed in this area allow you to adjust the size of the graph portion that scrolls off to the left when the display reaches the right edge, the initial time interval that the graph is to span, and the frequency with which the display is updated.
Data Collector Options	The options listed in this area control the collection of data. The sample interval indicates the amount of time to wait between recorded samples. The history depth indicates the number of samples to retain in the data-collector buffer. The save mode indicates the format in which the data-collector data is to be saved, either in binary or ASCII format.

Graph Tool Tools Menu

Use the **Tools** menu to bring up other tools. This menu provides the following options.

Option	Use
Query Tree	This option starts a query-tree tool. For more information, see "Query-Tree Tool" on page 11-19.
Status	This option starts a status tool. For more information, see "Status Tool" on page 11-20.
Disk Activity	This option starts a disk-activity tool. For more information, see "Activity Tools" on page 11-21.
Session Activity	This option starts a session-activity tool. For more information, see "Activity Tools" on page 11-21.

(1 of 2)

Option	Use
Disk Capacity	This option starts a disk-capacity tool. For more information, see "Activity Tools" on page 11-21.
Physical Processor Activity	This option starts a physical-processor tool. For more information, see "Activity Tools" on page 11-21.
Virtual Processor Activity	This option starts a virtual-processor tool. For more information, see "Activity Tools" on page 11-21.

(2 of 2)

Changing the Scale of Metrics

When **onperf** displays metrics, it automatically adjusts the scale of the y-axis to accommodate the largest value. You can use the Customize Metric dialog box (see "Graph Tool View Menu" on page 11-13) to establish one for the current display.

Displaying Recent-History Values

The **onperf** utility allows you to scroll back over previous metric values that are displayed in a line graph. This feature allows you to analyze a recent trend. The time interval to which you can scroll back is the *lesser* of the following intervals:

- The time interval over which the metric has been displayed
- The history interval specified in the graph-tool Configuration dialog box

  For more information, see "Graph Tool Configure Menu and the Configuration Dialog Box" on page 11-14.

Figure 11-11 illustrates the maximum scrollable intervals for metrics that span different time periods.

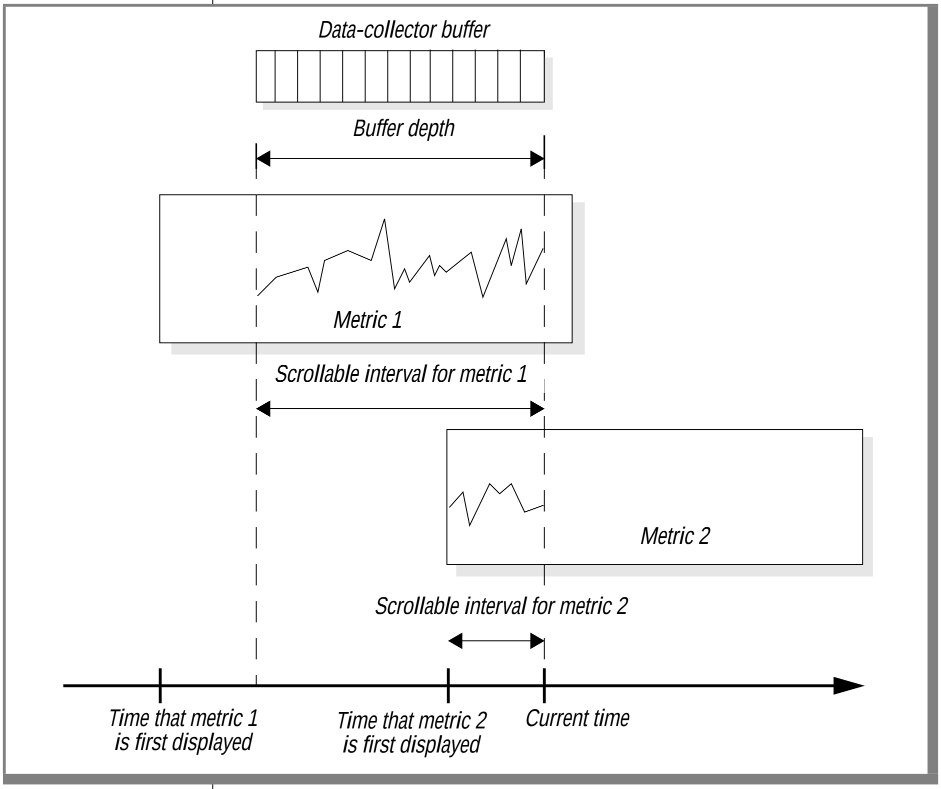

Data-collector buffer
Buffer depth
Metric 1
Scrollable interval for metric 1
Metric 2
Scrollable interval for metric 2
Time that metric 1 is first displayed
Time that metric 2 is first displayed
Current time

Figure 11-11
Maximum Scrollable Intervals for Metrics That Span Different Time Periods

Query-Tree Tool

The query-tree tool allows you to monitor the performance of individual queries. It is a separate executable tool that does not use the data-collector process. You cannot save query-tree tool data to a file. Figure 11-12 shows a diagram of the query-tree tool.

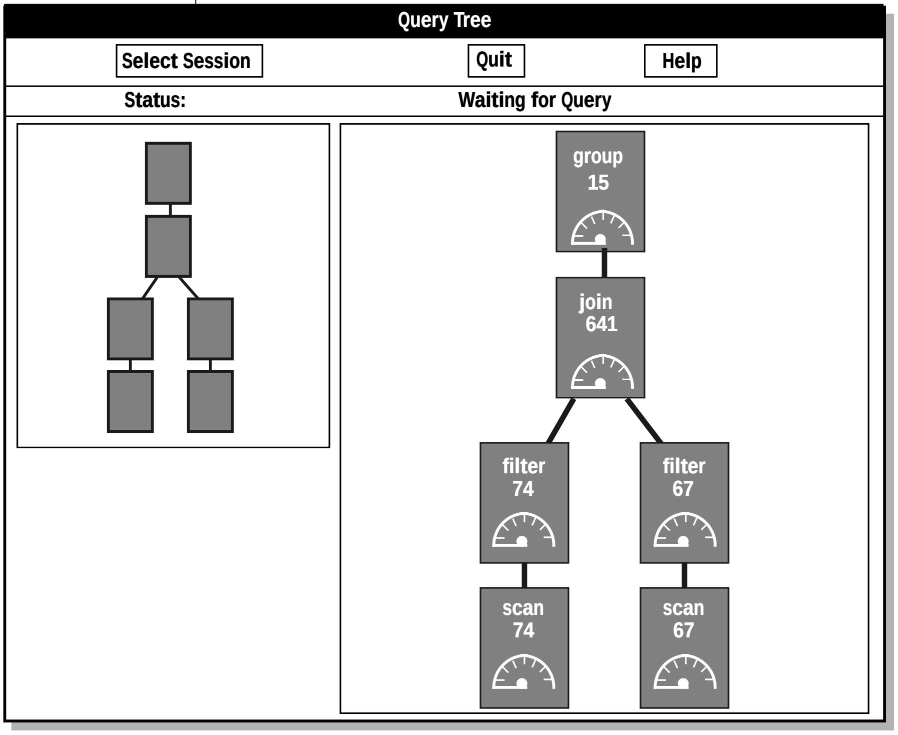


Figure 11-12
Query Tree Tool Window

This tool has a **Select Session** button and a **Quit** button. When you select a session that is running a query, the large detail window displays the SQL operators that constitute the execution plan for the query. The query-tree tool represents each SQL operator with a box. Each box includes a dial that indicates rows per second and a number that indicates input rows. In some cases, not all the SQL operators can be represented in the detail window. The smaller window shows the SQL operators as small icons.

The **Quit** button allows you to exit from the query-tree tool.

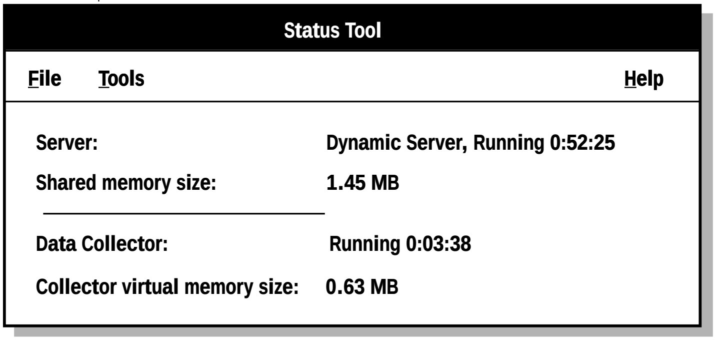
Status Tool

The status tool allows you to select metrics to store in the data-collector buffer. In addition, you can use this tool to save the data currently held in the data-collector buffer to a file. Figure 11-13 shows a status tool.

The status tool displays:

- the length of time that the data collector has been running.
- the size of the data-collector process area, called the *collector virtual memory size*.

  When you select different metrics to store in the data-collector buffer, you see different values for the collector virtual memory size.



Server: Dynamic Server, Running 0:52:25
Shared memory size: 1.45 MB

Data Collector: Running 0:03:38
Collector virtual memory size: 0.63 MB

Figure 11-13
Status Tool Window

Status Tool File Menu

The status tool **File** menu provides the following options.

Option	Use
Close	This option closes the tool. When it is the last remaining tool of the **onperf** session, **Close** behaves exactly like **Exit**.
Exit	This option exits **onperf**.

Status Tool Tools Menu

The **Tools** menu in the status tool is similar to the **Tools** menu in the graph tool, which is described in "Graph Tool Tools Menu" on page 11-16.

Activity Tools

Activity tools are specialized forms of the graph tool that display instances of the specific activity, based on a ranking of the activity by some suitable metric. You can choose from among the following activity tools:

- The disk-activity tool, which displays the top 10 activities ranked by total operations
- The session-activity tool, which displays the top 10 activities ranked by ISAM calls plus PDQ calls per second
- The disk-capacity tool, which displays the top 10 activities ranked by free space in megabytes
- The physical-processor-activity tool, which displays all processors ranked by nonidle CPU time
- The virtual-processor-activity tool, which displays all VPs ranked by VP user time plus VP system time

The activity tools use the bar-graph format. You cannot change the scale of an activity tool manually; **onperf** always sets this value automatically.

Activity Tool Graph Menu

The **Graph** menu provides you with options for closing, printing, and exiting the activity tool.

Activity Tool Tools Menu

The **Tools** menu is identical to the one that appears when you choose **Graph→Tools**. (For more information, see "Graph Tool Tools Menu" on page 11-16.)

Ways to Use onperf

The following sections describe different ways to use the **onperf** utility.

Routine Monitoring

You can use the **onperf** utility to facilitate routine monitoring. For example, you can display several metrics in a graph-tool window and run this tool throughout the day. Displaying these metrics allows you to monitor Dynamic Server performance visually at any time.

Diagnosing Sudden Performance Loss

When you detect a sudden performance dip, it is useful to examine the recent history of the Dynamic Server metrics values to identify any trend. The **onperf** utility allows you to scroll back over a time interval, as explained in "Displaying Recent-History Values" on page 11-17.

Diagnosing Performance Degradation

Performance problems that gradually develop might be difficult to diagnose. For example, if you detect a degradation in Dynamic Server response time, it might not be obvious from looking at the current metrics which value is responsible for the slowdown. The performance degradation might also be sufficiently gradual that you cannot detect a change by observing the recent history of metric values. To allow for comparisons over longer intervals, **onperf** allows you to save metric values to a file, as explained in "Status Tool" on page 11-20.

The onperf Metrics

The following sections describe the various metric classes. Each section indicates the scope levels available and describes the metrics within each class.

Dynamic Server performance depends on many factors, including your operating-system configuration, your Dynamic Server configuration, and your workload. It is difficult to describe relationships between **onperf** metrics and specific performance characteristics.

The approach taken here is to describe each metric without speculating on what specific performance problems it might indicate. Through experimentation, you can determine which metrics best monitor performance for a specific Dynamic Server instance.

Database Server Metrics

The scope for these metrics is always the named database server, which means the database server as a whole, rather than a component of the database server or disk space.

Metric Name	Description
CPU System Time	System time, as defined by the platform vendor
CPU User Time	User time, as defined by the platform vendor
Percent Cached (Read)	Percentage of all read operations that are read from the buffer cache without requiring a disk read, calculated as follows: `100 * ((buffer_reads — disk_reads) / (buffer_reads))`
Percent Cached (Write)	Percentage of all write operations that are buffer writes, calculated as follows: `100 * ((buffer_writes — disk_writes) / (buffer_writes))`
Disk Reads	Total number of read operations from disk
Disk Writes	Total number of write operations to disk

(1 of 3)

Metric Name	Description
Page Reads	Number of pages transferred to disk
Page Writes	Number of pages read from disk
Buffer Reads	Number of reads from the buffer cache
Buffer Writes	Number of writes to the buffer cache
ISAM Calls	Number of calls received at the ISAM layer of the database server
ISAM Reads	Number of read calls received at the ISAM layer of the database server
ISAM Writes	Number of write calls received at the ISAM layer of the database server
ISAM Rewrites	Number of rewrite calls received at the ISAM layer of the database server
ISAM Deletes	Number of delete calls received at the ISAM layer of the database server
ISAM Commits	Number of commit calls received at the ISAM layer of the database server
ISAM Rollbacks	Number of rollback calls received at the ISAM layer of the database server
Table Overflows	Number of times that the tblspace table was unavailable (overflowed)
Lock Overflows	Number of times that the lock table was unavailable (overflowed)
User Overflows	Number of times that the user table was unavailable (overflowed)
Checkpoints	Number of checkpoints written since Dynamic Server shared memory was initialized
Buffer Waits	Number of times that a thread waited to access a buffer
Lock Waits	Number of times that a thread waited for a lock
Lock Requests	Number of times that a lock was requested

(2 of 3)

Metric Name	**Description**
Deadlocks	Number of deadlocks detected
Deadlock Timeouts	Number of deadlock timeouts that occurred (Deadlock timeouts involve distributed transactions.)
Checkpoint Waits	Number of checkpoint waits; in other words, the number of times that threads have waited for a checkpoint to complete
Index to Data Pages Read-aheads	Number of read-ahead operations for index keys
Index Leaves Read-aheads	Number of read-ahead operations for index leaf nodes
Data-path-only Read-aheads	Number of read-ahead operations for data pages
Latch Requests	Number of latch requests
Network Reads	Number of ASF messages read
Network Writes	Number of ASF messages written
Memory Allocated	Amount of Dynamic Server virtual address space in kilobytes
Memory Used	Amount of Dynamic Server shared memory in kilobytes
Temp Space Used	Amount of shared memory allocated for temporary tables in kilobytes
PDQ Calls	The total number of parallel-processing actions that the database server performed
DSS Memory	Amount of memory currently in use for decision-support queries

(3 of 3)

Disk-Chunk Metrics

The disk-chunk metrics take the pathname of a chunk as the metric scope.

Metric Name	Description
Disk Operations	Total number of I/O operations to or from the indicated chunk
Disk Reads	Total number of reads from the chunk
Disk Writes	Total number of writes to the chunk
Free Space (MB)	The amount of free space available in megabytes

Disk-Spindle Metrics

The disk-spindle metrics take the pathname of a disk device or operation-system file as the metric scope.

Metric Name	Description
Disk Operations	Total number of I/O operations to or from the indicated disk or cooked file
Disk Reads	Total number of reads from the disk or operating-system file
Disk Writes	Total number of writes to the disk or operating-system file
Free Space	The amount of free space available in megabytes

Physical-Processor Metrics

The physical-processor metrics take either a physical-processor identifier (for example, `0` or `1`) or `Total` as the metric scope.

Metric Name	**Description**
Percent CPU System Time	CPU system time for the physical processors
Percent CPU User Time	CPU user time for the physical processors
Percent CPU Idle Time	CPU idle time for the physical processors
Percent CPU Time	The sum of CPU system time and CPU user time for the physical processors

Virtual-Processor Metrics

These metrics take a virtual-processor class as a metric scope (`cpu`, `aio`, `kio`, and so on). Each metric value represents a sum across all instances of this virtual-processor class.

Metric Name	**Description**
User Time	Accumulated user time for a class
System Time	Accumulated system time for a class
Semaphore Operations	Total count of semaphore operations
Busy Waits	Number of times that virtual processors in class avoided a context switch by spinning in a loop before going to sleep
Spins	Number of times through the loop

(1 of 2)

Metric Name	Description
I/O Operations	Number of I/O operations per second
I/O Reads	Number of read operations per second
I/O Writes	Number of write operations per second

(2 of 2)

Session Metrics

These metrics take an active session as the metric scope.

Metric Name	Description
Page Reads	Number of pages read from disk on behalf of a session
Page Writes	Number of pages written to disk on behalf of a session
Number of Threads	Number of threads currently running for the session
Lock Requests	Number of lock requests issued by the session
Lock Waits	Number of lock waits for session threads
Deadlocks	Number of deadlocks involving threads that belong to the session
Deadlock timeouts	Number of deadlock timeouts involving threads that belong to the session
Log Records	Number of log records written by the session
ISAM Calls	Number of ISAM calls by session
ISAM Reads	Number of ISAM read calls by session
ISAM Writes	Number of ISAM write calls by session
ISAM Rewrites	Number of ISAM rewrite calls by session
ISAM Deletes	Number of ISAM delete calls by session
ISAM Commits	Number of ISAM commit calls by session

(1 of 2)

Metric Name	Description
ISAM Rollbacks	Number of ISAM rollback calls by session
Long Transactions	Number of long transactions owned by session
Buffer Reads	Number of buffer reads performed by session
Buffer Writes	Number of buffer writes performed by session
Log Space Used	Amount of logical-log space used
Maximum Log Space Used	High-water mark of logical-log space used for this session
Sequential Scans	Number of sequential scans initiated by session
PDQ Calls	Number of parallel-processing actions performed for queries initiated by the session
Memory Allocated	Memory allocated for the session in kilobytes
Memory Used	Memory used by the session in kilobytes

(2 of 2)

Tblspace Metrics

These metrics take a tblspace name as the metric scope. A tblspace name is composed of the database name, a colon, and the table name (*database:table*). For fragmented tables, the tblspace represents the sum of all fragments in a table. To obtain measurements for an individual fragment in a fragmented table, use the Fragment Metric class.

Metric Name	Description
Lock Requests	Total requests to lock tblspace
Lock Waits	Number of times that threads waited to obtain a lock for the tblspace
Deadlocks	Number of times that a deadlock involved the tblspace
Deadlock Timeouts	Number of times that a deadlock timeout involved the tblspace

(1 of 2)

Metric Name	Description
ISAM Reads	Number of ISAM read calls involving the tblspace
ISAM Writes	Number of ISAM write calls involving the tblspace
ISAM Rewrites	Number of ISAM rewrite calls involving the tblspace
ISAM Deletes	Number of ISAM delete calls involving the tblspace
ISAM Calls	Total ISAM calls involving the tblspace
Buffer Reads	Number of buffer reads involving tblspace data
Buffer Writes	Number of buffer writes involving tblspace data
Sequential Scans	Number of sequential scans of the tblspace
Percent Free Space	Percent of the tblspace that is free
Pages Allocated	Number of pages allocated to the tblspace
Pages Used	Number of pages allocated to the tblspace that have been written
Data Pages	Number of pages allocated to the tblspace that are used as data pages

(2 of 2)

Fragment Metrics

These metrics take the dbspace of an individual table fragment as the metric scope.

Metric Name	Description
Lock Requests	Total requests to lock fragment
Lock Waits	Number of times that threads have waited to obtain a lock for the fragment
Deadlocks	Number of times that a deadlock involved the fragment
Deadlock Timeouts	Number of times that a deadlock timeout involved the fragment

(1 of 2)

Metric Name	**Description**
ISAM Reads	Number of ISAM read calls involving the fragment
ISAM Writes	Number of ISAM write calls involving the fragment
ISAM Rewrites	Number of ISAM rewrite calls involving the fragment
ISAM Deletes	Number of ISAM delete calls involving the fragment
ISAM Calls	Total ISAM calls involving the fragment
Buffer Reads	Number of buffer reads involving fragment data
Buffer Writes	Number of buffer writes involving fragment data
Sequential Scans	Number of sequential scans of the fragment
Percent Free Space	Percent of the fragment that is free
Pages Allocated	Number of pages allocated to the fragment
Pages Used	Number of pages allocated to the fragment that have been written to
Data Pages	Number of pages allocated to the fragment that are used as data pages

(2 of 2)

Case Studies and Examples

This appendix contains a case study and several extended examples of performance-tuning methods described in this manual.

Case Study

The following case study illustrates a case in which the disks are overloaded. It shows the steps taken to isolate the symptoms and identify the problem based on an initial report from a user, and it describes the needed correction.

A database application that has not achieved the desired throughput is being examined to see how performance can be improved. The operating-system monitoring tools reveal that a high proportion of process time was spent idle, waiting for I/O. The Dynamic Server administrator has increased the number of CPU VPs to make more processors available to handle concurrent I/O. However, throughput does not increase, which indicates that one or more disks are overloaded.

To verify the I/O bottleneck, the Dynamic Server administrator must identify the overloaded disks and the dbspaces that reside on those disks.

To identify overloaded disks and the dbspaces that reside on those disks

1. Check the asynchronous I/O (AIO) queues using **onstat -g ioq**, which gives the result shown in Figure A-1.

```
AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
adt   0      0     0        0        0        0        0
opt   0      0     0        0        0        0        0
msc   0      0     0        0        0        0        0
aio   0      0     0        0        0        0        0
pio   0      0     1        1        0        1        0
lio   0      0     1      341        0      341        0
gfd   3      0     1      225        2      223        0
gfd   4      0     1      225        2      223        0
gfd   5      0     1      225        2      223        0
gfd   6      0     1      225        2      223        0
gfd   7      0     0        0        0        0        0
gfd   8      0     0        0        0        0        0
gfd   9      0     0        0        0        0        0
gfd  10      0     0        0        0        0        0
gfd  11      0    28    32693    29603     3090        0
gfd  12      0    18    32557    29373     3184        0
gfd  13      0    22    20446    18496     1950        0
```

Figure A-1
Display from onstat -g ioq Utility

The **maxlen** and **totalops** columns show significant results in Figure A-1:

- The **maxlen** column shows the largest backlog of I/O requests to accumulate within the queue. The last three queues are much longer than any other queue in this column listing.
- The **totalops** column shows 100 times more I/O operations completed through the last three queues than for any other queue in the column listing.

The **maxlen** and **totalops** columns indicate that the I/O load is severely unbalanced.

Another way to check I/O activity is to use **onstat -g iov**. This option gives a slightly less detailed display for all VPs.

2. Check the AIO activity for each disk device using **onstat -g iof**, as Figure A-2 shows.

Figure A-2
Display from onstat -g iof Utility

```
AIO global files:
gfd pathname          totalops  dskread dskwriteio/s
3 /dev/infx2               0        0        00.0
4 /dev/infx2               0        0        00.0
5 /dev/infx2               2        2        00.0
6 /dev/infx2             223        0      2230.5
7 /dev/infx4               0        0        00.0
8 /dev/infx4               1        0        10.0
9 /dev/infx4             341        0      3410.7
10 /dev/infx4              0        0        00.0
11 /dev/infx5          32692    29602    309067.1
12 /dev/infx6          32556    29372    318466.9
13 /dev/infx7          20446    18496    195042.0
```

This display indicates the disk device associated with each queue. Depending on how your chunks are arranged, several queues can be associated with the same device. In this case, the total activity for **/dev/infx2** is the sum of the **totalops** column for queues 3, 4, 5, and 6, which is 225 operations. This activity is still insignificant when compared with **/dev/infx5**, **/dev/infx6**, and **/dev/infx7**.

3. Determine the dbspaces that account for the I/O load using **onstat -d,** as shown in Figure A-3.

Figure A-3
Display from onstat -d Utility

```
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
c009ad00 1        1        1        1        N        informix rootdbs
c009ad44 2        2001     2        1        N T      informix tmp1dbs
c009ad88 3        1        3        1        N        informix oltpdbs
c009adcc 4        1        4        1        N        informix histdbs
c009ae10 5        2001     5        1        N T      informix tmp2dbs
c009ae54 6        1        6        1        N        informix physdbs
c009ae98 7        1        7        1        N        informix logidbs
c009aedc 8        1        8        1        N        informix runsdbs
c009af20 9        1        9        3        N        informix acctdbs
 9 active, 32 total

Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
c0099574 1   1   500000   10000    9100              PO-   /dev/infx2
c009960c 2   2   510000   10000    9947              PO-   /dev/infx2
c00996a4 3   3   520000   10000    9472              PO-   /dev/infx2
c009973c 4   4   530000   250000   242492            PO-   /dev/infx2
c00997d4 5   5   500000   10000    9947              PO-   /dev/infx4
c009986c 6   6   510000   10000    2792              PO-   /dev/infx4
c0099904 7   7   520000   25000    11992             PO-   /dev/infx4
c009999c 8   8   545000   10000    9536              PO-   /dev/infx4
c0099a34 9   9   250000  450000    4947              PO-   /dev/infx5
c0099acc 10  9   250000  450000    4997              PO-   /dev/infx6
c0099b64 11  9   250000  450000    169997            PO-   /dev/infx7
 11 active, 32 total
```

In the **Chunks** display, the **pathname** column indicates the disk device. The **chk/dbs** column indicates the numbers of the chunk and dbspace that reside on each disk. In this case, only one chunk is defined on each of the overloaded disks. Each chunk is associated with dbspace number 9.

The **Dbspaces** display shows the name of the dbspace that is associated with each dbspace number. In this case, all three of the overloaded disks are part of the **acctdbs** dbspace.

Although the original disk configuration allocated three entire disks to the **acctdbs** dbspace, the activity within this dbspace suggests that three disks are not enough. Because the load is about equal across the three disks, it does not appear that the tables are necessarily laid out badly or improperly fragmented. However, you could get better performance by adding fragments on other disks to one or more large tables in this dbspace or by moving some tables to other disks with lighter loads.

Index