Administrator's Guide

for Informix Extended Parallel Server

Version 8.3
December 1999
Part No. 000-6552

Published by Informix® Press

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

Documentation Team: Twila Booth, Diana Chase, Kathy Eckardt, Virginia Panlasigui, Judith Sherwood, Karen Smith, Liz Suto

List of Chapters

Table of Contents

Introduction

In This Introduction

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

About This Manual

This manual contains important information on administering Informix Extended Parallel Server and includes information about how to install, configure, administer, and use the database server. It describes features, database server concepts, and procedures for performing database server management tasks.

A companion volume, the *Administrator's Reference*, contains reference material for using Informix database servers. If you need to tune the performance of your database server, see your *Performance Guide*.

Types of Users

This manual is written for the following users:

- Database users
- Database administrators
- Database server administrators
- Performance engineers
- Application developers

This manual assumes that you have the following background:

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

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

Software Dependencies

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

Assumptions About Your Locale

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

The default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms.

This locale supports U.S. English format conventions for dates, times, and currency, and also supports the ISO 8859-1 or Microsoft 1252 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

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

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

Demonstration Databases

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

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

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

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

New Features

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

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

- Configuration enhancements
- Performance enhancements
- New SQL functionality
- Year 2000 compliance
- Logical-log records
- Utility features
- Version 8.3 features from Informix Dynamic Server 7.30

Configuration Enhancements

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

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

Performance Enhancements

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

- Dynamic lock allocation
- Fuzzy checkpoints
- Skipping logical-log replay during restore
- Thread suspension to prevent database server failure for severe errors

Your *Performance Guide* describes additional performance improvements to Version 8.3 of the database server.

New SQL Functionality

You can now load and unload simple large objects to external tables.

Year 2000 Compliance

This manual describes the following Year 2000 compliance features in Version 8.3 of Extended Parallel Server:

- **DBCENTURY** environment variable
- Support for ISM, Version 2.0

Utility Features

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

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

Version 8.3 Features from Version 7.30

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

- Deferred constraints for all constraint types
- Memory-resident tables
- **ondblog** to change database logging
- Violations table

Documentation Conventions

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

The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Sample-code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** *`italics`*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

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

Icon Conventions

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

Comment Icons

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

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

Feature, Product, and Platform Icons

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

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

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

Sample-Code Conventions

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

```
CONNECT TO stores_demo
...

DELETE FROM customer
    WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

Additional Documentation

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

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

On-Line Manuals

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

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

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

Printed Manuals

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

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

Error Message Documentation

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

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

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

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

Documentation Notes, Release Notes, Machine Notes

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

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

On-Line File	Purpose
ADMINDOC_8.3	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
SERVERS_8.3	The release notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
XPS_*x.y*	The machine notes file describes any special actions that you must take to configure and use Informix products on your computer. The machine notes are named for the product described.

Related Reading

For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started* manual.

Compliance with Industry Standards

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

Informix Welcomes Your Comments

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

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

Send electronic mail to us at the following address:

doc@informix.com

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

We appreciate your suggestions.

The Database Server

Section I

Chapter

Introducing the Database Server

1

In This Chapter

This chapter introduces Informix Extended Parallel Server and discusses who uses the database server.

Database Server Users

The question "What is the database server?" means different things to different users. The following types of individuals who interact with the database server each have a different perspective:

- End users
- Application developers
- Database administrators
- Database server administrators
- Database server operators

End Users

End users use the *Structured Query Language* (SQL), often embedded in a client application, to access, insert, update, and manage information in databases. These end users might be completely unaware that they are using a database server. To them, the database server is a nameless aspect of the system that they are using.

Application Developers

For the developers of client applications, the database server offers a number of possibilities for data management, isolation levels, and so on.

For features and functions that the database server provides for developers, refer to the *Informix ESQL/C Programmer's Manual*. Other volumes, the *Informix Guide to SQL: Reference* and the *Informix Guide to SQL: Syntax*, also provide information that is useful for application developers.

Database Administrators

The *database administrator* (DBA) is primarily responsible for managing access control for a database, as described in "Database Server Security" on page 1-17. The DBA uses SQL statements to grant and revoke privileges to ensure that the correct individuals are able to perform the actions they need and that untrained or unscrupulous users are kept from performing potentially damaging or inappropriate resource-intensive activities. The *Informix Guide to SQL: Tutorial* and the *Informix Guide to SQL: Syntax* are also of interest to the DBA.

Database Server Administrators

Unlike the DBA, a database server administrator is responsible for maintenance, administration, and operation of the entire database server, which might manage many individual databases. For a description of the tasks involved in database server administration, refer to Chapter 2, "Overview of Database Server Administration."

Database Server Operators

Database server operators are responsible for routine tasks associated with database server administration, including backing up and restoring databases. The same person might fill the roles of administrator and operator.

Extended Parallel Server

Extended Parallel Server provides the following features:

- Client/server architecture
- Dynamic Scalable Architecture
- Multithreaded operation
- Scalability of performance
- Client/server operations
- Dynamic shared-memory management
- Direct (unbuffered) disk management
- Fault tolerance and high availability
- Database server security
- Year 2000 compliance

Extended Parallel Server supports the following types of data:

- Built-in data types
- Storage and access of simple large objects

Extended Parallel Server extends the following features for the management of large databases on multiple coservers:

- Extended parallel-processing architecture, which provides these features:
  - Extended multithreaded operation
  - Extended scalability of performance
  - Extended client/server operations
  - Extended dynamic shared-memory management
- Single point of administration

The following sections explain each of these features.

Client/Server Architecture

The *database server* provides reliable access to a database for client application programs. A *client* is an application program that a user runs to request information from a database that the database server manages.

When a client application connects to the database server and requests information, the database server locates the requested data within its databases and sends back the results. Depending on the type of request that the client application issues, the database server can return selected rows from tables within the databases that it manages, add or delete rows, or update particular columns within selected rows.

Client applications use Structured Query Language (SQL) to send requests for data to the database server. Client applications can use the data that the database server returns in a variety of ways, including simple displays on computer screens or complicated reports. Client programs include the DB-Access utility and programs that you write with an Informix application programming interface (API) such as Informix ESQL/C, Informix ODBC Driver, C++, or Java.

The database server performs additional activities such as coordinating concurrent requests from multiple clients and enforcing physical and logical consistency on the data to provide reliable access to the database.

The database server administrator starts the database server processes. After the database server is initialized, database server processes run continuously during the period that users access data. For more information on starting the database server, refer to Chapter 10, “Initializing the Database Server.”

Client Application Types

Two broad classes of applications operate on data that is stored 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 usually 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 usually multiuser applications with acceptable response times measured in fractions of a second.

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 has been captured through OLTP operations over time. These applications provide information that is often used for accounting, strategic planning, and decision-making. Data within the database is usually queried but not updated during DSS operations. Typical DSS applications include payroll, inventory, and financial reports.

A recent approach to DSS consolidates enterprisewide data in a separately designed environment, commonly called a *data warehouse*. A data warehouse stores business data for a company in a single, integrated relational database that provides a historical perspective on information for DSS applications.

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

Decision-support applications have the following characteristics:

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

Connection to a Database Server

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

At the source-code level, a client application connects to the database server through an SQL statement. Beyond that, the client use of connection facilities is transparent to the application. Library functions that are automatically included when a client program is compiled enable the client to connect to the database server.

As the database administrator, you specify the database server or coserver names and types of connections that can be made. The connectivity information is in the **sqlhosts** file.

To connect to the database server, the client specifies a database server name, which is a logical name assigned by the database administrator. The DBA associates this name with physical characteristics stored in **sqlhosts**, such as the host name, connection type and port number.

For a description of these connection features, refer to Chapter 6, "Client / Server Communications."

Dynamic Scalable Architecture

Informix database servers implement an advanced architecture that Informix calls *Dynamic Scalable Architecture* (DSA). DSA provides distinct performance advantages for both single-processor and symmetric multiprocessor computers. These advantages, which this section describes further, are as follows:

- A small number of database server processes can service a large number of client application processes, with the following benefits:
  - Reduced operating-system overhead (fewer processes to run)
  - Reduced overall memory requirements
  - Reduced contention for resources within the database management system (DBMS)
- DSA provides more control over setting priorities and scheduling database tasks than the operating system does.

Parallel-Processing Architecture

The multithreaded database server exploits *symmetric multiprocessor* (SMP) and uniprocessor architectures.

DSA particularly exploits SMP and *massively parallel processing* (MPP) computer systems. DSA provides the following additional benefits on those systems:

- Multiple processes can work in parallel for one client.
- On some multiprocessor computers, you can bind database server processes to specific CPUs.

Figure 1-1 shows the three major components of the Informix database server architecture.

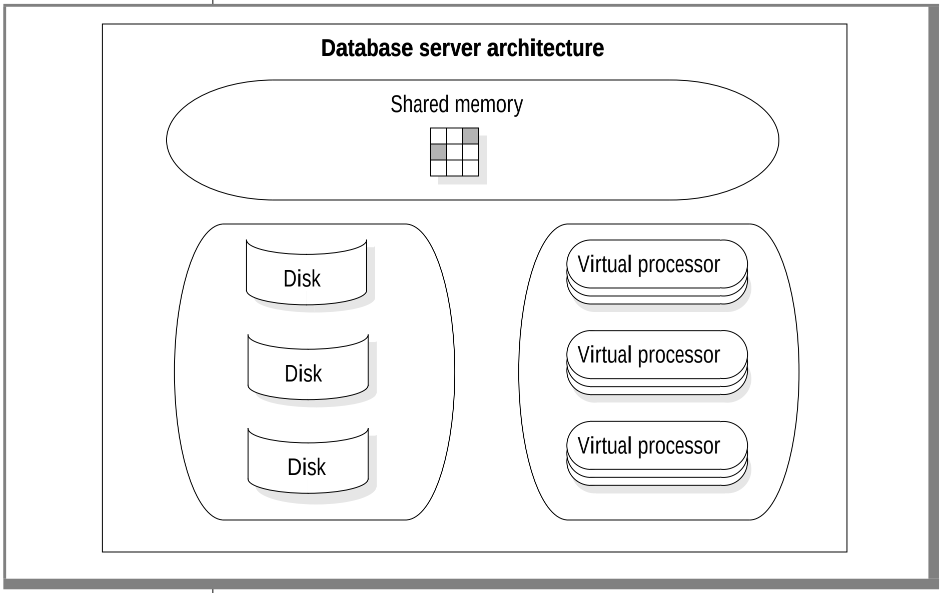


Figure 1-1
Database Server Architecture Components

Informix database server architecture includes the following the three major components:

- Virtual processors

  A virtual processor is a task that the operating system schedules for execution on the CPU. A database server thread is a task that the virtual processor schedules internally for processing. Database server virtual processors are *multithreaded* because they run multiple concurrent threads. For more information, refer to Chapter 11, "Virtual Processors and Threads."

- Shared memory

  Shared memory consists of a resident and virtual portion. The database server uses the resident portion to cache data from the disk for faster access by multiple client applications. The database server uses the virtual portion to maintain and control the resources that virtual processors require and to read in large blocks of infrequently accessed data. For more information on how the database server uses shared memory, see Chapter 13, "Shared Memory," and Chapter 14, "Managing Shared Memory."

- Disk component

  The disk component is a collection of one or more units of disk space assigned to the database server. All the data in the databases and all the system information necessary to maintain the database server system reside within the disk component. For more information on the disk components, see Chapter 15, "Data Storage."

The multithreaded database server manages access to one or more relational databases for client applications. In a relational database, data is organized in tables that consist of rows and columns.

Scalability

DSA allows the 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, and networking functions in parallel.

For more information on virtual processors, refer to "Virtual Processors" on page 11-3. To understand how the database server manages shared memory to scale performance, see Chapter 13, "Shared Memory." For tuning and performance information, refer to your *Performance Guide*.

High Performance

The database server achieves high performance through the following mechanisms:

- Dynamic shared-memory management
- Raw disk management
- Dynamic thread allocation
- Parallelism

The following sections explain each of these mechanisms.

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. This sharing of data in memory prevents redundant disk I/O and the corresponding degradation in performance that might otherwise occur.

Database server shared memory contains both data from the database and control information. Because the data needed by various applications is located in a single, shared portion of memory, all control information needed to manage access to that data can be located in the same place. The database server adds memory dynamically as needed, and as the administrator, you can also add segments to shared memory if necessary. For information about adding a segment to shared memory, refer to Chapter 14, "Managing Shared Memory."

Direct Disk Access

The database server uses direct, or *unbuffered,* disk access to improve the speed and reliability of disk I/O operations. When you assign disk space to the database server, you can bypass the file-buffering mechanism that the operating system provides. The database server itself manages the data transfers between disk and memory.

UNIX provides unbuffered disk access by means of character-special devices (also known as *raw* disk devices). For more information about character-special devices, refer to your UNIX documentation.

When you store tables on raw disks or 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.

If performance is not a primary concern, you can configure the database server to store data in regular (buffered) operating-system files, which are also known as *cooked* files. When the database server uses cooked files, it manages the file contents, but the operating system manages the disk I/O.

For more information about how the database server uses disk space, see Chapter 15, "Data Storage."

Dynamic Thread Allocation

The database server supports multiple client applications using 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 flexible database server architecture provides dynamic load balancing for both on-line transaction processing (OLTP) and decision-support applications.

For a description of database server threads, refer to Chapter 11, "Virtual Processors and Threads."

Fragmentation and Parallelism

The database server uses local table partitioning (also called *fragmentation*) to intelligently distribute tables across disks for better performance. For very large databases (VLDBs), the ability to fragment data is important to manage the data efficiently.

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

PDQ is most effective when you use it with fragmentation. For an overview of fragmentation and PDQ, refer to Chapter 17, "Table Fragmentation and PDQ."

Fault Tolerance

The database server uses the following logging and recovery mechanisms to protect data integrity and consistency in the event of an operating-system or media failure:

- Storage-space and logical-log backups
- External backup and restore
- Fast recovery
- Point-in-time restore
- Restartable restore
- Mirroring
- Data replication

Storage Space and Logical-Log Backups of Transaction Records

The database server manages data in logical storage units called storage spaces. The most common storage space is the *dbspace,* in which the database server stores traditional data such as integer, decimal and floating-point numbers, fixed-length or variable-length character strings, and so forth.

The database server stores transaction records and changes to the database server in *logical-log files.* You can back up storage spaces and logical-log files while users are accessing databases.

You can also create incremental backups on-line. Incremental backups enable you to back up only data that has changed since the last backup, which reduces the amount of time required to back up and restore your data and logical logs.

After a media failure, if critical data was not damaged (and the database server remains in on-line mode), you can restore only the data that was on the failed media, leaving other data available during the restore.

If the failure causes the database server to go off-line, you can restore all data, including the critical data and the root dbspace. An off-line restore is called a *cold restore*.

For more information on backup and restore, refer to the *Backup and Restore Guide*.

Backup Verification

If you use ON-Bar as your backup tool, you might want to verify the completeness and consistency of a storage-space backup. After you successfully verify a storage-space backup, you can restore it safely. If ON-Bar indicates problems with the backup, contact Informix Technical Support. The *Backup and Restore Guide* explains how to use ON-Bar to verify backups.

On Extended Parallel Server, you can issue a single command that verifies the data after it is backed up.

External Backup and Restore

An external backup allows you to make copies of disks that contain storage spaces without using ON-Bar. Later on, you can use an external restore to restore these disks to the database server without using ON-Bar or the Informix Storage Manager. External backups are especially useful if your site has special hardware or software that allows rapid copying of data directly to and from your primary data disks. The *Backup and Restore Guide* explains external backup and restore.

Fast Recovery

When the database server starts up, it checks if the physical log is empty because that implies that it shut down in a controlled fashion. If the physical log is *not* empty, the database server automatically performs an operation called *fast recovery*. Fast recovery automatically restores databases to a state of physical and logical consistency after a system failure that might have left one or more transactions uncommitted. During fast recovery, the database server uses its *logical log* and *physical log* to perform the following operations:

- Restore the databases to their state at the last checkpoint
- Roll forward all committed transactions since the last checkpoint
- Roll back any uncommitted transactions

The database server spawns multiple threads to work in parallel during fast recovery. For a detailed explanation of fast recovery, refer to Chapter 24, "Checkpoints and Fast Recovery."

Point-in-Time Restore

After the time of the backup, you can restore data from backup media to a specified point in time. This feature enables you to restore a corrupted database to a point at which you know that the data was reliable. For more information, refer to the *Backup and Restore Guide*.

Mirroring

When you use disk mirroring, the database server writes data to two locations. Mirroring eliminates data loss due to storage device failures. If mirrored data becomes unavailable for any reason, the mirror of the data is available immediately and transparently to users.

The database server relies on the operating system for bad-sector mapping. When the database server confirms the failure of a disk, it suspends I/O operations on that chunk. If the chunk has been mirrored, the database server directs I/O requests to the mirror. If an unmirrored chunk containing critical information fails, the database server shuts down immediately. Critical information includes logical-log files, the physical log, and the root dbspace. The chunks on which this data resides are referred to as *critical dbspaces*.

Important: *Informix recommends that you mirror critical dbspaces that contain logical-log files, the physical log, and the root dbspace.*

For more information about mirroring and critical data, refer to Chapter 25, "Mirroring."

Database Server Security

The database server enforces access privileges to databases and tables through the use of the SQL statements GRANT and REVOKE. For more information about database and table privileges, refer to the *Informix Guide to Database Design and Implementation* and the *Informix Guide to SQL: Syntax*.

Year 2000 Compliance

The database server and client products support the **DBCENTURY** environment variable for Year 2000 compliance. The **DBCENTURY** environment variable lets you choose the appropriate expansion (present, past, closest, or future century) for DATE and DATETIME values that have only a one- or two-digit year. For more information, see the *Informix Guide to SQL: Reference*.

Extended Parallel-Processing Architecture

The parallel-processing architecture provides high performance for database operations on computing platforms that range from a computer with a single CPU to parallel-processing platforms composed of dozens or hundreds of computers.

A parallel-processing platform is a set of independent computers that operate in parallel and communicate over a high-speed network, bus, or interconnect. The database server can run on several types of parallel-processing platforms, including:

- Massively parallel processing (MPP) systems composed of multiple computers that are connected to a single high-speed communication subsystem
- Clusters of stand-alone computers that are connected over a high-speed network
- Symmetric multiprocessing (SMP) computers

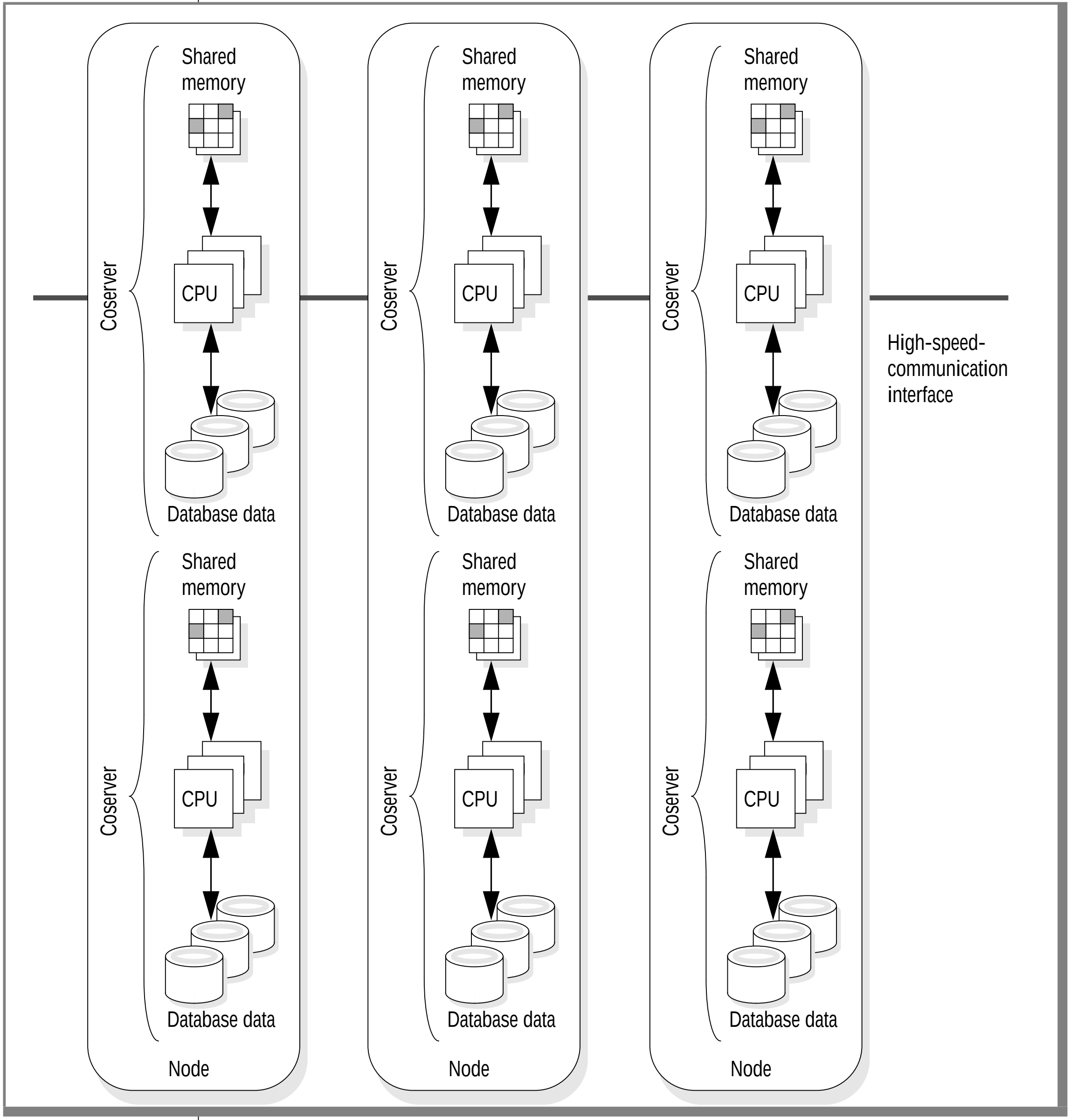
An individual computer that operates within a parallel-processing platform is referred to as a *node*. A node can be a uniprocessor or an SMP computer. Within a computer of this type, resources can be segregated into smaller independently addressed subsystems with separate CPUs, memory regions, and I/O channels. Independent subsystems configured within an SMP computer are considered to be nodes for purposes of database server configuration.

You can configure the database server on a single computer or a parallel-processing platform as a set of one or more coservers. A *coserver* is the functional equivalent of a database server that operates on a single node. Each coserver performs database operations in parallel with the other coservers that make up the database server. Each coserver independently manages its own resources and activities such as logging, recovery, locking, and buffers.

The database server does not require that any hardware resources be shared between nodes. This approach is sometimes referred to as a *shared-nothing* architecture.

Figure 1-2 illustrates the database server parallel-processing architecture.

Figure 1-2
Database Server in a Shared-Nothing Environment



Extended Multithreaded Operation

Each coserver within the database server is multithreaded. This multithreaded architecture allows each coserver to provide the highest degree of parallelism. The database server extends this parallelism across coservers by allowing tables to be fragmented across multiple coservers and by providing parallel execution of database operations across multiple coservers.

Parallel Execution Within a Coserver

Within each coserver, the database operations are performed in a multithreaded fashion. Queries are broken down into component steps that can often be performed in parallel within a given coserver.

Each coserver uses a relatively small number of processes called virtual processors to support multiple client applications or large DSS queries. 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. For more information about multithreaded execution within a coserver, refer to Chapter 11, “Virtual Processors and Threads.”

The database server provides for parallel execution both within coservers and between coservers. You can spread the data from a given table across multiple disks managed by multiple coservers. This feature is known as fragmentation. Spreading data across multiple disks allows coservers to perform disk I/O operations in parallel, thereby accessing multiple table fragments simultaneously.

Parallel Execution on Multiple Coservers

Multiple threads associated with a query can run simultaneously on multiple coservers, and on single coservers that are running on SMP computers. This capability is referred to as the parallel database query (PDQ) feature. Together, fragmentation and PDQ allow the database server to perform parallel operations on tables.

Although a coserver can perform disk I/O only on the table fragments that it manages, each coserver accesses its own disks independently and in parallel with other coservers. Data managed by a particular coserver is accessed only by that coserver. This arrangement ensures that coservers can run with the highest degree of parallelism in a shared-nothing environment.

For instance, if your database server is configured with 4 coservers and each coserver manages 6 disks, you could fragment data from a single table across all 24 disks. If each coserver runs on a separate SMP platform with at least 6 CPUs, the database server can take advantage of the shared-nothing architecture of each coserver to access all of the table fragments simultaneously for fully parallel processing. As your table grows, you can add more computers and configure more coservers to increase the capacity of the your database server. In this way, you can process larger and larger amounts of information in roughly the same time.

For a description of fragmented tables, refer to the *Informix Guide to Database Design and Implementation*. For information about using fragmentation and PDQ for maximum performance, see your *Performance Guide*.

Extended Scalability

Extended Parallel Server extends DSA to provide a high degree of scalability for decision-support applications and growing workloads. A key element for scalability is the parallel execution that the *parallel database query* (PDQ) and table fragmentation features provide. PDQ and table fragmentation can improve performance dramatically when the database server processes queries that access data which resides on disks located on different coservers.

For more information on these features, refer to Chapter 17, "Table Fragmentation and PDQ." For tuning and performance information, refer to your *Performance Guide*.

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

Extended Client/Server Operations

Extended Parallel Server extends the client / server architecture of Dynamic Server to the parallel-processing platforms and multiple coservers.

Connection Coserver

A *connection coserver* is the coserver that manages a client connection to a database server. The connection coserver determines if a client request can be satisfied with data that resides on that coserver alone or if it requires data that resides on other coservers.

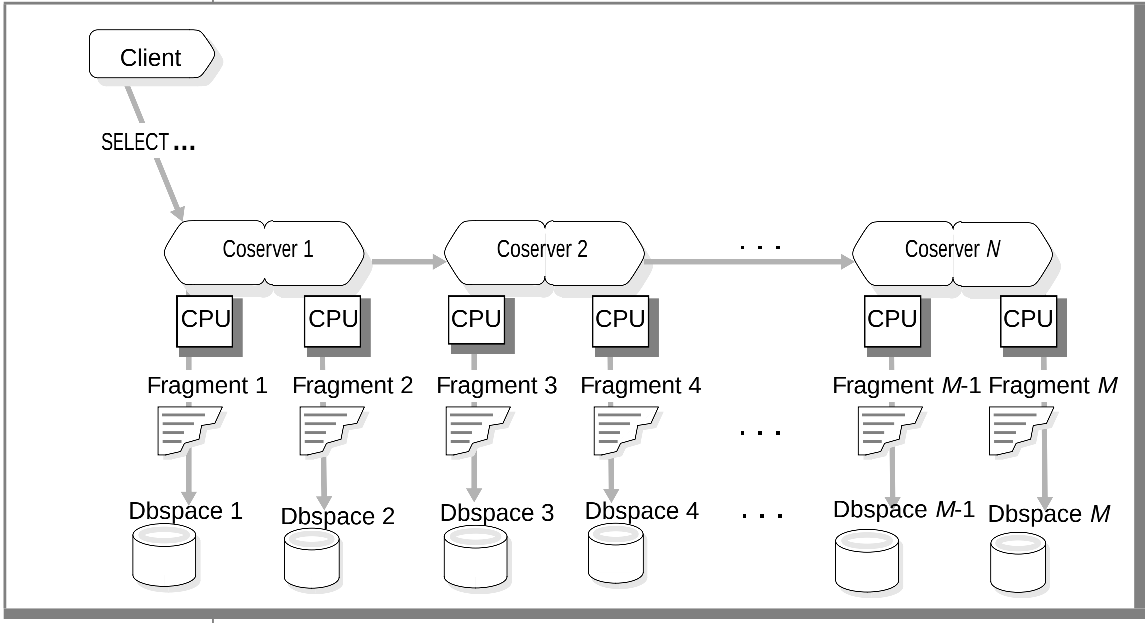
Different clients can connect to different coservers. For example, Client A can connect to **Coserver 1**, and Client B can connect to **Coserver 2**. **Coserver 1** is the connection coserver for Client A, and **Coserver 2** is the connection coserver for Client B.

Participating Coserver

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

Figure 1-3 shows a client that is accessing a very large database that is fragmented across many coservers. **Coserver 1** is the connection coserver. **Coservers 1** through ***N*** are all participating coservers.

Figure 1-3
Client Query That Coservers Service



On-Line Transaction Processing and Decision-Support Applications

Extended Parallel Server provides support for both OLTP and DSS applications.

DSS applications perform more complex tasks than OLTP, often including scans of entire tables, manipulation of large amounts of data, multiple joins, and the creation of temporary tables. Such operations involve many calculations and large amounts of memory. As a result, DSS execution times are typically far longer than OLTP execution times. However, Extended Parallel Server excels at handling the complex queries that are typical of decision-support applications.

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

Extended Parallel Server provides exceptional performance for DSS applications such as data marts and data warehouses.

For more information about data mart applications, refer to the *Informix Guide to Database Design and Implementation*.

Extended Parallel Server does not support distributed data queries where a single transaction contains queries to more than one database across multiple database servers.

Extended Dynamic Shared-Memory Management

All applications that use data from a particular coserver share data in the memory space of that coserver. After one application reads data from a table fragment that resides on a coserver, other applications can access whatever data is already in memory. This sharing of data in memory prevents redundant disk I/O and the corresponding degradation in performance that might otherwise occur.

Each database server or coserver adds memory dynamically as needed, and you, as the administrator, can also add segments to shared memory if necessary. For information on how to add a segment to database server shared memory, refer to Chapter 13, “Shared Memory.”

Direct Disk Access

Extended Parallel Server uses the same direct, or *unbuffered*, disk access as Dynamic Server to improve the speed and reliability of disk I/O operations.

For more information about how the database server uses disk space, refer to Chapter 15, “Data Storage.”

Single Point of Administration

With Extended Parallel Server, many coservers function together to form a single database server. These coservers usually have the same number of CPUs and equal amounts of memory and disk storage space. This uniformity makes it easier to administer all coservers from a single point.

The following features help you control and manage multiple coservers:

- Centralized configuration file
- Coserver groups (cogroups)
- Dbslices
- Logslices
- Centralized message log
- Coordinated data-dictionary cache
- Centralized database server utilities

Centralized Configuration

The database server has one centralized configuration file that applies to all coservers. To simplify the task of configuring multiple database server instances, Extended Parallel Server uses global configuration parameters for information that applies to all coservers. You specify these global configuration parameters once in the global section of the configuration file. The configuration file also contains a section for information that applies to individual coservers. For more information about global and coserver-specific configuration parameters, see "Configuring Multiple Coservers" on page 5-3.

Coserver Groups

A coserver group (referred to as a *cogroup* for convenience) is a subset of the coservers within the database server. You can create cogroups to simplify system and database administration. A coserver can be in more than one cogroup. For example, you might create the following cogroups.

Cogroup	Purpose
CUSTOMER_COGROUP	For the coservers that own the fragmented data for a customer application
ACCOUNT_COGROUP	For the coservers that own the fragmented data for an accounting application

Extended Parallel Server provides a system-defined cogroup called **cogroup_all**. The **cogroup_all** cogroup includes all of the coservers that your ONCONFIG configuration file defines.

Cogroups help you administer a large number of coservers. For more details see Chapter 10, “Initializing the Database Server.”

Dbslices

A *dbslice* is a named set of dbspaces that usually spans multiple coservers. A dbslice is managed as a single storage object. For more information about dbslices, refer to Chapter 15, “Data Storage.”

Logslices

A *logslice* is a set of logical-log files that occupy a dbslice and are owned by multiple coservers, one logical-log file per dbspace. Logslices simplify the process of adding and deleting logical-log files by treating sets of them as single entities. For more information about logslices, see “Logslices” on page 20-5.

Centralized Message Log

To facilitate administration, Informix recommends that you use a centralized message-log file. To do so, place the message-log file in a file system that is accessible by all coservers. All coservers can then write their messages to this file. To identify the coserver that generated a given message, check the coserver number that begins each message.

Avoid placing message-log files in local file systems. That arrangement complicates administration of the database server by scattering log messages across multiple files. This arrangement also prevents you from viewing messages in sequence.

Centralized Database Server Utilities

Extended Parallel Server provides centralized command-line and graphical monitoring utilities.

The onutil Utility

The **onutil** utility allows you to monitor and manage tables and other database objects across multiple coservers. For more information about **onutil**, refer to the utilities chapter in the *Administrator's Reference*.

The xctl Utility

The **xctl** utility allows you to run coserver-specific utilities on multiple coservers and execute operating-system commands on multiple nodes. You can include a coserver-specific utility such as **oninit**, **onlog**, or **onstat**, or an operating-system command in an **xctl** command-line, along with syntax to designate the nodes or coservers on which the command is to run. For more information on **xctl**, refer to the utilities chapter in the *Administrator's Reference*.

Coordinated Data-Dictionary Cache

When a table is fragmented across coservers, each table fragment is owned by the coserver on which it resides. Only the owning coserver can update the system catalog tables that define the table fragment.

When a session executes an SQL statement that accesses data on a table fragment that is located on another coserver, the database server reads the system catalog tables from a participating coserver and stores them in structures that it can access more efficiently. These structures are created in the virtual portion of shared memory for use by all sessions on that coserver. These structures constitute the data-dictionary cache for a coserver.

Each coserver maintains its own data-dictionary cache. Each coserver can cache the data-dictionary information retrieved from other coservers that own a particular database definition. The database server ensures that each coserver accesses the most current system tables in its data-dictionary cache.

Only one **INFORMIXDIR** directory and one logical **sqlhosts** file exist across all coservers.

Control of Resources in Parallel

Each coserver owns and manages a set of dbspaces and the table fragments that reside on those dbspaces. Each coserver also manages its own logging, recovery, locking, and buffer management for the table fragments that it owns.

Each coserver manages the database objects that it owns and coordinates activities with the other coservers. The database server provides the following services to coordinate parallel processing among coservers:

- The request manager
- The query optimizer
- The data-dictionary manager
- The scheduler

Request Manager

A request manager resides on each coserver. The request manager decides how to divide a query and how to distribute the workload to balance the tasks across coservers. The request manager works with other database server services to determine if a client request should involve multiple coservers:

- The query optimizer to determine the best way of executing a request
- The data-dictionary manager to determine where the data resides
- The scheduler to distribute the request

The request manager on the coserver where the user connects is called the *connection* request manager for that user session. If a request involves execution on multiple coservers, the connection request manager passes context information to the other coservers so that they can establish local session context for the user session.

Query Optimizer

The query optimizer decides how to perform a query. The query optimizer first finds all feasible query plans. A query plan is a distinct method of executing a query that takes into account the order in which tables are read, how they are read (by index or sequentially), and how they are joined with other tables in the query. The query optimizer assigns a cost to each component operation that is required under each plan and then selects the plan with the lowest cost for execution.

The query optimizer uses pertinent information from the data-dictionary manager to determine the degree of parallelism of the request.

Data-Dictionary Manager

The data-dictionary manager contains information about data definitions, the system catalogs that contain them, and the coservers on which those system catalogs reside. The Data-Dictionary Manager provides coservers with access to system catalogs that reside on other coservers. Once a coserver has obtained a data definition, that coserver can retain a copy in local shared memory. The only coserver that can modify a data definition is the one on which the appropriate system catalog resides.

Each coserver manages and maintains the table fragments that reside on its dbspaces but is aware of the table fragments on other coservers through the Data-Dictionary Manager located on every coserver. If a coserver requires access to table fragments that it does not own, the coserver sends a request to the coserver that owns the dbspace to retrieve it.

Scheduler

A scheduler resides on each coserver. The scheduler distributes execution tasks to the other coservers. The scheduler activates a part of the execution plan on each coserver to allocate the pertinent resources.

Chapter

2 Overview of Database Server Administration

In This Chapter

As a database server administrator, you need to be aware of the tasks and responsibilities that fall into your domain. This chapter describes the three types of tasks that you need to perform as an administrator of an Informix database server:

- Initial installation and configuration tasks
- Routine tasks that you perform on a regular basis
- Configuration tasks that you perform less frequently

For a summary of database server administration tasks, refer to "Summary of Administration Tasks" on page 2-8.

Database Server Administrator

Most database server administrative tasks require you to have the privileges accorded to the database server administrator. On UNIX, you must log in as user **informix** to acquire the privileges of the database server administrator.

Initial Tasks

When you first acquire the database server, you need to perform some initial installation and configuration tasks. For a description of these tasks, see Chapter 3, "Installing and Configuring the Database Server."

If you are moving from one version of the database server to another, refer to the *Informix Migration Guide*.

You must also configure connectivity for your database server and client applications, as explained in Chapter 6, "Client / Server Communications."

These tasks can seem complicated and time consuming. Fortunately, they are not common tasks and are not representative of most of the administrative work that the database server requires.

Configuration Tasks

Configuration tasks are generally either setup tasks that involve initiating functionality or maintenance and performance-adjustment tasks that are required by the usage pattern of your database server.

Managing Disk Space

You are responsible for planning and implementing the layout of information managed by the database server on disks. The way you distribute the data can greatly affect the performance of the database server.

Chapter 15, "Data Storage," explains the advantages and drawbacks of different disk configurations. Chapter 16, "Managing Disk Space," describes the actual disk-management tasks.

If you plan to use more than one database server instance on the same computer, refer to Chapter 7, "Multiple Residency," to familiarize yourself with the issues of multiple residency.

For information on how to fragment tables and indexes over multiple disks to improve performance, concurrency, data availability, and backups, refer to Chapter 17, "Table Fragmentation and PDQ," and to your *Performance Guide*.

Managing Database-Logging Status

You can specify whether the default logging mode for databases is buffered or unbuffered and whether the logging mode is ANSI compliant.

Databases always use transaction logging. However, you can specify logging or nonlogging tables in your database.

For information about these logging options, refer to Chapter 18, "Logging." For information on how to change logging options, refer to Chapter 19, "Managing Database-Logging Status."

Managing the Logical Log

Although backing up logical-log files is a routine task, logical-log administration (placing and sizing log files and specifying high-water marks) is required even when none of your databases use transaction logging. For information about logical-log administration, refer to Chapter 20, "Logical Log."

For instructions on creating and modifying the logical-log configuration, see Chapter 21, "Managing Logical-Log Files."

For information on backing up the logical log, see the *Backup and Restore Guide*.

Managing the Physical Log

You can change the size and location of the physical log as part of effective disk management. For more information about the physical log, refer to Chapter 23, "Managing the Physical Log."

Using Mirroring

For information on mirroring, refer to Chapter 25, "Mirroring." Informix recommends that you mirror at least your root dbspace. For instructions on tasks related to mirroring, see Chapter 26, "Using Mirroring."

Managing Shared Memory

Managing shared memory includes the following tasks:

- Changing the size or number of buffers (by changing the size of the logical-log or physical-log buffer, or changing the number of buffers in the shared-memory buffer pool)
- Changing shared-memory parameter values, if necessary
- Changing forced residency (on or off, temporarily or for this session)
- Tuning checkpoint intervals
- Adding segments to the virtual portion of shared memory

For information on how the database server uses shared memory, refer to Chapter 13, "Shared Memory." For a description of how to manage shared memory, refer to Chapter 14, "Managing Shared Memory."

Managing Virtual Processors

The configuration and management of virtual processors (VPs) has a direct impact on the performance of a coserver or database server. The optimal number and mix of VPs for your database server depends on your hardware and on the types of applications that your database server supports.

For an explanation of virtual processors, refer to Chapter 11, "Virtual Processors and Threads." For information on the procedures to manage virtual processors, refer to Chapter 12, "Managing Virtual Processors."

Managing Parallel Database Query

You can control the resources that the database uses to perform decision-support queries in parallel. You need to balance the requirements of decision-support queries against those of on-line transaction processing (OLTP) queries. The resources that you need to consider include shared memory, threads, temporary table space, and scan bandwidth. For information on parallel database query (PDQ) and how fragmentation affects the performance of PDQ, refer to your *Performance Guide*.

Routine Tasks

Depending on the needs of your organization, you might be responsible for performing the periodic tasks described in the following paragraphs. Not all of these tasks are appropriate for every installation. For example, if your database server is available 24 hours a day, 7 days a week, you might not bring the database server to off-line mode, so database server operating mode changes would not be a routine task.

Changing Modes

The database server administrator is responsible for starting up and shutting down the database server by changing the mode. Chapter 9, "Managing Database Server Operating Modes," explains how to change database server modes.

Backing Up Data and Logical-Log Files

To ensure that you can recover your databases in the event of a failure, Informix recommends that you make frequent backups of your storage spaces and logical logs.

How often you back up the storage spaces depends on how frequently the data is updated and how critical it is. A backup schedule might include a complete (level-0) backup once a week, incremental (level-1) backups daily, and level-2 backups hourly. You also need to perform a level-0 backup after performing administrative tasks such as adding a dbspace or logical log or enabling mirroring.

Back up each logical-log file as soon as it fills. You can back up these files manually or automatically.

For information on using ON-Bar, see the *Backup and Restore Guide*.

Monitoring Database Server Activity

The Informix database server design lets you monitor every aspect of the database server. "Sources of Information for Monitoring" on page 2-11 provides descriptions of the available information, instructions for obtaining information, and suggestions for its use. As a result of monitoring, you might need to change your configuration in one of the ways described in "Summary of Administration Tasks" in the following section.

Checking for Consistency

Informix recommends that you perform occasional checks for data consistency. For a description of these tasks, refer to Chapter 27, "Consistency Checking."

Summary of Administration Tasks

The following tables summarizes the initial tasks, routine tasks, and configuration tasks that you perform to administer the database server. Figure 2-1 lists initial administration tasks.

Figure 2-1
Initial Tasks Summary

Initial Tasks	Reference
Plan for database server	"Planning for the Database Server" on page 3-4
Preinstallation operating-system changes	"Configuring the Operating System" on page 3-6
Install the database server	*Installation Guide*
Configure the database server	Chapter 5, "Configuring the Database Server"
Define connectivity	Chapter 6, "Client/Server Communications"
Initialize the database server	Chapter 10, "Initializing the Database Server"

Figure 2-2 lists database server configuration tasks.

Figure 2-2
Configuration Tasks Summary

Configuration Tasks	**Reference**
Manage disk space and storage spaces	Chapter 16, "Managing Disk Space"
Loading external tables	*Administrator's Reference*
Analyze disk configuration	Chapter 15, "Data Storage"
Changing database-logging status	Chapter 18, "Logging" Chapter 19, "Managing Database-Logging Status"
Administer the logical log:	
■ Size and placement of logical logs	Chapter 20, "Logical Log"
■ Modify logical log configuration	Chapter 21, "Managing Logical-Log Files"
Administer the physical log	Chapter 23, "Managing the Physical Log" Chapter 24, "Checkpoints and Fast Recovery"
Manage mirroring	Chapter 25, "Mirroring" Chapter 26, "Using Mirroring"
Manage shared memory	Chapter 13, "Shared Memory" Chapter 14, "Managing Shared Memory"
Manage virtual processors	Chapter 12, "Managing Virtual Processors"
Manage Parallel Data Query (PDQ)	*Performance Guide*
Migrate to a different database server version	*Informix Migration Guide*

Figure 2-3 lists routine administration tasks.

Figure 2-3
Routine Tasks Summary

Routine Tasks	Reference
Change database server modes	Chapter 9, "Managing Database Server Operating Modes"
Refresh data warehousing tables	*Administrator's Reference*
Back up data and logical-log files:	
■ ON-Bar	*Backup and Restore Guide*
■ Informix Storage Manager	*Informix Storage Manager Administrator's Guide*
Monitor activity	Monitoring information in relevant chapters of this book
Check for data consistency	Chapter 27, "Consistency Checking"
Tune database server performance	*Performance Guide*

Monitoring Database Server Activity

You can monitor information about the following aspects of the database server:

- Configuration
- Checkpoints
- Shared memory
  - Shared-memory segments
  - Shared-memory profile
  - Buffers
  - Latches
  - Locks
- Active tblspaces
- Virtual processors
- Sessions and threads

- Transactions
- Parallel database queries (PDQ) and resources
- Databases
- Logging activity
  - Logical-log files
  - Physical-log file
  - Physical-log and logical-log buffers
  - Log backup status
- Disk usage
  - Chunks
  - Tblspaces and extents
  - Simple large objects in dbspaces

Sources of Information for Monitoring

You can gather information about database server activity from the following sources:

- Message log
- Event alarm
- System console
- SMI tables
- **onstat** utility
- **onutil** CHECK options

The following sections explain each of these sources.

Message Log

The database server *message log* is an operating-system file. The messages contained in the database server message log do not usually require immediate action. To report situations that require your immediate attention, the database server uses the event-alarm feature. See "Event Alarm" on page 2-13. To specify the message-log pathname, set the MSGPATH configuration parameter. For more information about MSGPATH, see the chapter on configuration parameters in the *Administrator's Reference*.

Monitoring the Message Log

Informix recommends that you monitor the message log once or twice a day to ensure that processing is proceeding normally and that events are being logged as expected. Use the **onstat -m** command to obtain the name of the message log and the 20 most-recent entries. Use a text editor to read the complete message log. Use an operating-system command (such as the UNIX command **tail -f)** to see the messages as they occur. For a list of these messages, see the chapter on message-log messages in the *Administrator's Reference*.

Monitor the message-log size as well because the database server appends new entries to this file. Edit the log as needed, or back it up to tape and delete it.

If the database server experiences a failure, the message log serves as an audit trail for retracing the events that develop later into an unanticipated problem. Often the database server provides the exact nature of the problem and the suggested corrective action in the message log.

You can read the database server message log for a minute-by-minute account of database server processing in order to catch events before a problem develops. However, Informix does not expect you to perform this kind of monitoring.

Changing the Destination for Message-Log Messages

You can change the value of MSGPATH while the database server is in on-line mode, but the changes do not take effect until you reinitialize shared memory.

Event Alarm

The database server provides a mechanism for automatically triggering administrative actions based on an event that occurs in the database server environment. This mechanism is the event-alarm feature.

To use the event-alarm feature, set the ALARMPROGRAM configuration parameter to the full pathname of an executable file that performs the necessary administrative actions.

For more information, see the appendix on event alarms and the chapter on configuration parameters in the *Administrator's Reference*.

System Console

The database server sends messages that are useful to the database server administrator by way of the *system console*. To specify the destination pathname of console messages, set the CONSOLE configuration parameter. For more information about CONSOLE, see the chapter about configuration parameters in the *Administrator's Reference*.

You can change the value of CONSOLE while the database server is in on-line mode, but the changes do not take effect until you reinitialize shared memory.

SMI Tables

The *system-monitoring interface* (SMI) tables are special tables managed by the database server that contain dynamic information about the state of the database server. You can use SELECT statements on them to determine almost anything you might want to know about your database server. For a description of the SMI tables, see the chapter about the **sysmaster** database in the *Administrator's Reference*.

onstat Utility

The **onstat** utility provides a way to monitor database server shared memory from the command line. The **onstat** utility reads data from shared memory and reports statistics that are accurate for the instant during which the command executes. That is, **onstat** provides information that changes dynamically during processing, including changes in buffers, locks, and users.

One useful feature of **onstat** output is the heading that indicates the database server status. Whenever the database server is blocked, **onstat** displays the following line after the banner line:

```
Blocked: reason
```

The variable *reason* can take one of the following values.

Reason	Description
CKPT	Checkpoint
LONGTX	Long transaction
ARCHIVE	Ongoing archive
MEDIA_FAILURE	Media failure
HANG_SYSTEM	Database server failure
DBS_DROP	Dropping a dbspace
DDR	Discrete high-availability data replication
LBU	Logs full high-water mark

For an example of what **onstat** displays when the database server is blocked to preserve logical-log space for administrative tasks, see "Monitoring the Logical Log for Fullness" on page 21-16.

onutil CHECK Utility

The **onutil** utility provides a number of CHECK commands for monitoring database server information regarding tables, indexes, and availability of disk space. For more information about **onutil**, see the utilities chapter in the *Administrator's Reference*.

xctl Utility

The **xctl** utility allows you to run coserver-specific utilities on multiple coservers and execute operating-system commands on multiple nodes. You can include a coserver-specific utility such as **oninit**, **onlog**, or **onstat**, or an operating-system command in an **xctl** command-line, along with syntax to designate the nodes or coservers on which the command is to run. For more information about **xctl**, see the utilities chapter in the *Administrator's Reference*.

Configuration

Section II

Chapter

3 Installing and Configuring the Database Server

In This Chapter

Configuring a database-management system requires many decisions, such as where to store the data, how to access the data, and how to protect the data. How you install and configure the database server can greatly affect the performance of database operations.

You can customize the database server so that it functions optimally in your particular data-processing environment. For example, a database server instance that serves 1000 users who execute frequent, short transactions is much different from the database server instance in which a few users make long and complicated searches.

This chapter has three purposes:

- To introduce files, environment variables, and utilities that the database server uses for configuration
- To let you quickly start the database server with a simple configuration
- To provide a foundation for more detailed information that appears throughout this book

The chapter also introduces terminology and covers some of the issues that you must consider before you install the database server. The following topics appear in this chapter:

- Planning for the database server
- Configuring the operating system
- Allocating disk space
- Installing the database server
- Setting environment variables
- Configuring connectivity information
- Preparing the ONCONFIG configuration file

- Starting and administering the database server
- Monitoring configuration information

Planning for the Database Server

When you are planning for your database server, consider your priorities and your environment.

Considering Your Priorities

As you prepare the initial configuration and plan your backup strategy, keep in mind the characteristics of your database server:

- Do applications on other computers use this database server instance?
- What is the maximum number of users that you can expect?
- How much help or supervision will the users require?
- To what extent do you want to control the environment of the users?
- Are you limited by resources for space, CPU, or availability of operators?
- How much does the database server instance need to do without supervision?
- Does the database server usually handle many short transactions or fewer long transactions?

Considering Your Environment

Before you start the initial configuration of your database server, collect as much of the following information as possible. You might need the assistance of your system administrator to obtain this information:

- The host names and IP addresses of the other computers on your network

  Does your platform support the Network Information Service (NIS)?

- The host names and Internet protocol (IP) addresses of the nodes on your parallel-processing platform

  For more information about parallel-processing platform, refer to "Setting Up Node Names" on page 3-7.

- The disk-controller configuration

  How many disk drives are available? Are some of the disk drives faster than others? How many disk controllers are available? What is the disk-controller configuration?

- What are the requirements, features, and limitations of your storage manager and backup devices?

  For more information, see the *Informix Storage Manager Administrator's Guide* or your storage-manager documentation.

- Operating-system shared memory and other resources

  How much shared memory is available? How much of it can you use for the database server?

  The machine notes file indicates which parameters are applicable for each UNIX platform.

Configuring the Operating System

Before you can start configuring the database server, you must configure the operating system appropriately. You might need the assistance of the system administrator for this task.

Modifying UNIX Kernel Parameters

The machine notes file contains recommended values that you use to configure operating-system resources. Use these recommended values when you configure the operating system on each node within your parallel-processing platform.

If the recommended values for the database server differ significantly from your current environment, consider modifying your operating-system configuration. For more information, see your *Performance Guide*.

On some operating systems, you can specify the amount of shared memory allocated to the database server. The amount of available memory influences the values that you can choose for the shared-memory parameters in your configuration file. In general, increasing the space available for shared memory enhances performance. You might also need to specify the number of locks and semaphores.

For background information on the role of the UNIX kernel parameters, see Chapter 14, "Managing Shared Memory."

Preparing the Operating System for the Database Server

You must perform additional tasks to prepare your operating system for the database server. You might need the assistance of the system administrator in order to perform these tasks:

- Set up node names
- Understand operating system administration facilities

Setting Up Node Names

The hardware architecture of your computer platform and the intended use of your database server both play a role in determining the appropriate number of coservers that you configure.

A basic rule of thumb for configuring coservers is one coserver per node. However, the following exceptions to this rule apply to the following types of parallel-processing platforms:

- If your platform consists of only one computer with between eight and ten (depending on the platform) or fewer CPUs, you can choose the single-coserver configuration option for ease of administration.
- If your platform is an SMP computer with more than eight or ten CPUs, or if the physical memory available on the computer is more than double the size of the virtual address space of a single process, you can improve performance by configuring multiple coservers.
- If your platform is an SMP computer that can be partitioned into independent computing subsystems, you can achieve both ease of administration and improved performance by partitioning the computer and configuring a single coserver per subsystem. In this case, each subsystem is regarded as an individual node for configuration purposes.

For more information about configuring coservers, refer to Chapter 5, "Configuring the Database Server."

Your system administrator specifies the node names and the IP addresses in your operation system **hosts** file on each node. Informix suggests that you share the same **hosts** file across all of the nodes on your platform. For more information about this file, see "Connectivity Files" on page 6-13.

If the operating system requires it, the system administrator defines one set of host names for network access and another set of host names for access through the high-speed communication interface or network. Use the appropriate host names for high-speed communication between nodes in the NODE configuration parameter in your ONCONFIG file. For a description, see "Host Name Field" on page 6-26.

Using Operating-System Administration Facilities

Each operating system has its own set of tools for administration of multiple nodes and users. For example, these tools include a central console and possibly the **kerberos** utility or various platform support programs. Your operating-system administration guide indicates which of these tools are applicable for database server system administration on your hardware platform.

If your operating system uses a central console, you can use that console to manage and maintain all of the nodes on that platform. Some operating systems refer to the console as a central-control workstation. Other operating systems refer to it as an administrative workstation.

Allocating Disk Space

Configuring your disks is the most important task for obtaining optimum performance with data marts and data warehouses. Disk I/O is the longest portion of the response time for an SQL operation. The database server offers parallel access to multiple disks on a computer.

Extended Parallel Server offers the advantage of parallel access to multiple disks spread across many coservers.

Before you allocate the disk space, study the information about disk space in your operating-system administration guide.

To allocate disks for the database server

1. Configure a raw device or create an unbuffered file for each disk.
2. Create standard device names or filenames.
3. Set permissions, ownership, and group for each raw device or unbuffered file.
4. Set up access to disks across all nodes.

Creating a Raw Device or Unbuffered File

To achieve better performance, UNIX uses raw disk devices, and Windows NT uses unbuffered NTFS files to bypass the buffering of disk I/O that the operating system normally performs. To create raw devices on UNIX, follow the instructions provided with your operating system. To configure unbuffered NTFS files on Windows NT, follow the steps in the installation program.

Some operating systems use the concept of a *logical volume,* and others use a *logical unit*. Each of these terms represents the basic unit of physical disk space that the operating system reserves for the database server. A chunk is a physical partition, logical volume, a logical unit, or regular file that has been assigned to the database server. Use the **onutil** utility to create chunks on a coserver.

Important: *You must define your logical volume or logical unit to be no larger than a chunk. To determine which chunk size your operating system supports, refer to your machine notes file.*

Creating Standard Device Names

Informix recommends that you use symbolic links to assign abbreviated standard device names for each raw disk device. If you have symbolic links, you can replace a disk that has failed with a new disk by assigning the symbolic name to the new disk.

To create a link between the character-special device name and another filename, use the UNIX link command (usually **ln)**.

Execute the UNIX command **ls -l** (**ls -lg** on BSD) on your device directory to verify that both the devices and the links exist. The following example shows links to raw devices. If your operating system does not support symbolic links, you can use hard links.

```
% ls -lg
crw-rw---  /dev/rxy0h
crw-rw---  /dev/rxy0a
lrwxrwxrwx /dev/my_root@->/dev/rxy0h
lrwxrwxrwx /dev/raw_dev2@->/dev/rxy0a
```

Extended Parallel Server requires standard device names across all coservers on UNIX. Use standard naming conventions for the chunk paths.

Setting Permissions, Ownership, and Group

Files or raw devices that the database server uses must have the appropriate ownership and permissions.

On UNIX, the owner and group must be **informix**, and the permissions must be set to read and write for both user and group (but not for others).

If you want users other than **informix** or **root** to execute ON-Bar commands, create a **bargroup** group. Only members of **bargroup** can execute ON-Bar commands. The **bargroup** group is not created automatically during database server installation. For instructions on creating a group, see your UNIX documentation.

Setting Up Disk Access Across Nodes

The file system on which the **INFORMIXDIR** directory is installed should be exported to and mounted by all nodes that are defined for the database server. In addition, you must copy the following utilities on each node:

- **oninit**
- **onmode**
- **onstat**

Place the directory that contains these copied utilities before **$INFORMIXDIR/etc** in the search path because the **INFORMIXDIR** directory might be on another node.

Installing the Database Server

Installation refers to the process of loading files from the product distribution media onto your computer and running the installation script to set up the product files correctly. For information about how to install the database server, see your *Installation Guide*.

Warning: *Do not try to install multiple copies of the database server from the distribution media on the same computer.*

Using Multiple Residency

If you run more than one instance of the database server on the same computer, it is called *multiple residency*. To prepare to use multiple instances of the database server, first install and configure *one* database server. To prepare for multiple residency, initialize multiple instances of the database server. For more information, refer to Chapter 8, "Using Multiple Residency."

Upgrading the Database Server

If you are upgrading your database server from an earlier version, see the *Informix Migration Guide* for instructions.

Setting Environment Variables

To start, stop, or access a database server, you must set the appropriate environment variables.

You can include the environment variables in the operating-system boot-up procedure, include the environment variables in your **.cshrc** or **.informix** file, or set the environment variables each time you log on. For information about **.cshrc**, refer to your operating-system manuals. For information about the **.informix** file and others files that the database server uses, refer to the *Administrator's Reference*.

Tip: *The "Informix Guide to SQL: Reference" contains a complete list of environment variables.*

Required Environment Variables

You must set the following environment variables before you access the database server or perform most administrative tasks:

- **INFORMIXDIR**

  The **INFORMIXDIR** environment variable specifies the directory where the product files are installed. Set **INFORMIXDIR** to the directory where you installed your Informix database server.

- **PATH**

  The **PATH** environment variable specifies the location of executable files (**$INFORMIXDIR/bin**).

- **ONCONFIG**

  The **ONCONFIG** environment variable specifies the name of the active ONCONFIG configuration file. The next section describes how to prepare the ONCONFIG file. After you prepare the ONCONFIG file, set the **ONCONFIG** environment variable to the name of the file.

  Users running client applications do not need to set the **ONCONFIG** environment variable.

  If the **ONCONFIG** environment variable is not present, the database server uses configuration values from the file **$INFORMIXDIR/etc/onconfig**.

- **INFORMIXSERVER**

  The **INFORMIXSERVER** environment variable specifies the name of the default database server. It can have one of the following values:

  - The same value as specified for the DBSERVERNAME or DBSERVERALIASES configuration parameter in the ONCONFIG file
  - A specific coserver name, using the following format:

    ```
    dbservername.coserver_number
    ```

    In the coserver name, *dbservername* is the value that is assigned to the **DBSERVERNAME** configuration parameter in the ONCONFIG file, and *coserver_number* is the value assigned to the COSERVER configuration parameter for the connection coserver.

GLS

Global Language Support

The GLS feature allows you to create databases that use the diacritics, collating sequence, and monetary and time conventions of the language that you select. To use GLS, you must set appropriate environment variables. Users might need additional environment variables, such as **LC_COLLATE**, to describe their environment fully. For a detailed discussion of GLS, refer to the *Informix Guide to GLS Functionality*.

Other Environment Variables

The **DBSPACETEMP** environment variable specifies dbspaces that the database server can use to store temporary tables for a particular session. Using **DBSPACETEMP** can improve performance.

DBSPACETEMP specifies the location of dbspaces or dbslices. If **DBSPACETEMP** is not set, the default location is NOTCRITICAL.

For further information about **DBSPACETEMP**, refer to the chapter about configuration parameters in the *Administrator's Reference*. For performance considerations, refer to your *Performance Guide*.

The **TERM**, **TERMCAP**, **TERMINFO** and **INFORMIXTERM** environment variables specify the type of terminal interface. The basic login procedure usually sets the **TERM** environment variable. This environment variable is not required for initialization, but it must be set before you can run an application. The other three environment variables might be required, depending on your environment. You might need assistance from the UNIX system administrator to set these variables because they are highly system dependent.

Environment Variable Files

To set additional environment variables for each database server user, the database administrator can prepare the **$INFORMIXDIR/etc/informix.rc** environment-configuration file.

To override environment variables that have been automatically set, users can use a private environment-variable file, **~/.informix**, or assign new values to environment variables individually.

Configuring Connectivity

The connectivity information allows a client application to connect to any Informix database server on the network. The connectivity data for a particular database server includes the database server name, the type of connection that a client can use to connect to it, the host name of the computer or node on which the database server runs, and the service name by which is known.

You must prepare the connectivity information even if the client application and the database server are on the same computer or node.

For detailed information about the connectivity information, see Chapter 6, "Client / Server Communications."

The sqlhosts File

The **sqlhosts** file contains connectivity information. The default location of this file is **$INFORMIXDIR/etc/sqlhosts**. If you store the information in another location, you must set the **INFORMIXSQLHOSTS** environment variable.

Preparing the ONCONFIG Configuration File

After the database server is installed, it must be configured before it can be brought on-line. *Configuration* refers to setting specific parameters that customize the database server for your data-processing environment: quantity of data, number of tables, types of data, hardware, number of users, and security needs.

Also consider the number of coservers in your system.

Defaults for most of the configuration parameters are set during the installation of the product. However, you can change the configuration parameters to customize the database server for your data-processing environment.

Chapter 4, "Configuration Parameters," provides an overview of the configuration parameters. For information about additional configuration parameters, see the chapter on configuration parameters in the *Administrator's Reference*.

The configuration parameters customize the database server for your data-processing environment. This manual refers to the file that stores the configuration parameters as the *ONCONFIG file*. You must prepare this file before you initialize the database server. After you prepare the configuration file, you must set the **ONCONFIG** environment variable to identify it to the database server.

Creating a Configuration File

The installation procedure loads templates for two configuration files in the **etc** subdirectory of the **INFORMIXDIR** directory: **onconfig.std** and **onconfig.xps**. For a listing of both files, see the appendix on database server files in the *Administrator's Reference*. For more information on the configuration parameters listed in **onconfig.std** and **onconfig.xps**, refer to Chapter 4, "Configuration Parameters."

To determine the template to use for your configuration file:

- For single coserver environments, use the **onconfig.std** template file.
- For multiple-coserver environments, use the **onconfig.xps** template file.

Important: *Do not modify the template files. The database server provides these files as templates and not as functional configuration files.*

Make a copy of the appropriate template file and set the **ONCONFIG** environment variable to refer to your copy.

Important: *If you omit parameters in your copy of the ONCONFIG file**,** the database server uses values in the* ***onconfig.std*** *file for the missing parameters during initialization.*

For the default values contained in the **onconfig.std**, refer to the configuration parameter chapter in the *Administrator's Reference*. For information on the order of files that the database server checks for configuration values, refer to "Process Configuration File" on page 10-5.

Creating an ONCONFIG File

To prepare the ONCONFIG file, use a text editor and complete the following steps:

1. Make a copy of a standard ONCONFIG file template:
   - For a multiple-coserver configuration, copy the following file:

     ```
     $INFORMIXDIR/etc/onconfig.xps
     ```
   - For all other configurations, copy the following file:

     ```
     $INFORMIXDIR/etc/onconfig.std
     ```

   Store the new file in the same directory as the template. You can give your new configuration file any name that meets the requirements of your operating system.
2. Edit your new ONCONFIG file to modify the configuration parameters that you decide to change.
3. Set the **ONCONFIG** environment variable to the name of your new ONCONFIG file.

Starting and Administering the Database Server

After you install and configure the database server, you need to perform one or more of the following steps:

- Prepare to connect to applications.
- Start the database server and initialize disk space.
- Create dbspaces.
- Perform administrative tasks.

Preparing to Connect to Applications

Application programs use the **sqlhosts** file to look up connectivity information. On UNIX, the **sqlhosts** file resides in the **$INFORMIXDIR/etc** directory. For more information about the **sqlhosts** file, refer to "The sqlhosts File" on page 6-21.

You do not need to specify all possible network connections in **sqlhosts** before you initialize the database server. But to make a new connection available, you must take the database server off-line and then bring it back to on-line mode once again.

Configuring the Database Server Page Size

Use the PAGESIZE configuration parameter to set the database server page size. You can specify a page size of 2048, 4096, or 8192 bytes. For more information, see the chapter on configuration parameters in the *Administrator's Reference*.

Steps for changing the page size

1. Unload your data using a page-independent tool such as external tables, **dbexport**, or the SQL statement UNLOAD.
2. Set the PAGESIZE value in the ONCONFIG file.
3. Reinitialize the database server with the new page size set.
4. Load your data back into the database.
5. Perform a level-0 backup of the system.

For information on unloading and loading data, see the *Informix Migration Guide*. For information on how to use external tables to unload, see the chapter on loading with external tables in the *Administrator's Reference*.

Starting the Database Server and Initialize Disk Space

To bring the database server to on-line mode, enter **oninit**.

If you are starting a new database server, use the **oninit** command with the **-i** flag to initialize the disk space and bring the database server to on-line mode.

If you are starting Extended Parallel Server for the first time, use the following command to initialize the disk space and to bring the database server into on-line mode on all coservers:

```
xctl -C oninit -iy
```

For more information on configuring Extended Parallel Server, see "Choosing a Coserver Configuration" on page 5-10.

For a description of the types of initialization and associated commands, refer to Chapter 9, "Managing Database Server Operating Modes."

Warning: *When you initialize disk space, all of the existing data in the database server is destroyed. Initialize disk space only when you are starting a new database server.*

Performing Administrative Tasks

After you initialize the database server, you need to perform the following administrative tasks:

- Prepare the operating-system scripts to automatically start and stop the database server.
- Make arrangements for backup management.
- Make sure that users have the correct environment variables.
- Warn the UNIX system administrator about **cron** jobs.

Preparing the Startup and Shutdown Scripts

You can modify the start-up script to initialize the database server automatically when your computer enters multiuser mode. You can also modify your shutdown script to shut down the database server in a controlled manner whenever your system shuts down.

Before starting the database server, your script should check to see if all nodes are available and that they can be reached.

To prepare the startup script, add UNIX and database server utility commands to the startup script so that the script performs the following steps.

To prepare the startup script

1. Set the **INFORMIXDIR** environment variable to the full pathname of the directory in which the database server is installed.
2. Set the **PATH** environment variable to include the **$INFORMIXDIR/bin** directory.
3. Set the **ONCONFIG** environment variable to the desired configuration file.
4. Set the **INFORMIXSERVER** environment variable so that the **sysmaster** database can be updated (or created, if needed).
5. Execute **oninit**, which starts the database server and leaves it in on-line mode.
6. If you plan to initialize multiple versions of the database server (multiple residency), you must reset **ONCONFIG** and **INFORMIX-SERVER** and re-execute **oninit** for each instance of the database server.
7. If you are using Informix Storage Manager (ISM) for managing database server backups, you must start the ISM server on each node. For information about how to start the ISM server, refer to your *Installation Guide*.

If different versions of the database server are installed in different directories, you must reset **INFORMIXDIR** and repeat the preceding steps for each different version.

To shut down the database server in a controlled manner whenever your system shuts down, add UNIX and database server utility commands to the shutdown script so that the script performs the following steps.

To prepare the shutdown script

1. Set the **INFORMIXDIR** environment variable to the full pathname of the directory in which the database server is installed.
2. Set the **PATH** environment variable to include the **$INFORMIXDIR/bin** directory.
3. Set the **ONCONFIG** environment variable to the desired configuration file.
4. Execute **onmode -ky**, which initiates Immediate-Shutdown and takes the database server off-line.

   If you are running multiple versions of the database server (multiple residency), you must reset **ONCONFIG** and re-execute **onmode -ky** for each instance.

If different versions of the database server are installed in different directories, you must reset **INFORMIXDIR** and repeat the preceding steps for each different version.

In the shutdown script, the database server shutdown commands should execute after all client applications have completed their transactions and exited.

Making Sure That Users Have the Correct Environment Variables

Make sure that every user of an Informix product has the correct environment variables. Each user must set the following environment variables before accessing the database server:

- **INFORMIXDIR**
- **INFORMIXSERVER**
- **PATH**

All users who use database server utilities such as **onstat** must set the **ONCONFIG** environment variable to the name of the ONCONFIG file.

Three techniques are available for setting **INFORMIXDIR**, **INFORMIX-SERVER**, **PATH**, and **ONCONFIG**:

- Ask the UNIX administrator to set these environment variables for every user during the login procedure.
- Modify the login procedures for each database server user so that these environment variables are set during login.
- Educate your users to set the environment variables manually every time that they want to work with the database server.

Users might need other environment variables, such as **TERMCAP** and **LC_COLLATE**, to describe their environment fully. You can prepare an environment-configuration file or assign values individually as discussed in "Other Environment Variables" on page 3-13. For a detailed discussion of environment variables, refer to the *Informix Guide to SQL: Reference*.

Warning UNIX System Administrator About cron Jobs

The database server creates the **.inf.*servicename*** and **VP.*servername.xxC*** files in the **/INFORMIXTMP** directory. Some UNIX systems run **cron** jobs that routinely delete all files from the **/INFORMIXTMP** directory. For information about files that the database server uses, refer to the *Administrator's Reference*.

Setting Up Your Storage Manager and Storage Devices

If you use ON-Bar as your backup tool, you must set up a storage manager and storage devices before you can back up and restore data. For information on ON-Bar and related configuration parameters, see the *Backup and Restore Guide*.

ON-Bar is packaged with Informix Storage Manager (ISM). The *storage manager* is an application that manages the storage devices and media that contain backups. The storage manager handles all media labeling, mount requests, and storage volumes. ISM can back up data to as many as four storage devices at a time. ISM stores data on simple tape drives, optical disk devices, and file systems. However, you can purchase a third-party storage manager if you want to use more sophisticated storage devices, backups to more than four storage devices at a time, or backups over a network.

When you plan your storage-space and logical-log backup schedule, make sure that the storage devices and backup operators are available to perform backups. For information about managing backup devices and media, refer to the *Informix Storage Manager Administrator's Guide* or to your third-party storage-manager documentation.

Creating Storage Spaces

After the database server is initialized, you can create storage spaces such as dbspaces and dbslices as desired. For a description of storage spaces, refer to "Logical Units of Storage" on page 15-14. For a discussion of the allocation and management of storage spaces, refer to Chapter 16, "Managing Disk Space."

If you know that your queries require temporary tables for sorts and hash joins, you might want to create temporary dbspaces or dbslices. For more information on query operations that require temporary tables, refer to "Temporary Tables" on page 15-30.

Monitoring Configuration Information

One of the tasks of the database server administrator is to keep records of the configuration. Methods of obtaining configuration information are as follows.

Using Command-Line Utilities

Use the following utilities to monitor configuration information.

onstat -c

Execute **onstat -c** to display a copy of the ONCONFIG file. For information about the ONCONFIG file, see the chapter on configuration parameters in the *Administrator's Reference*.

Changes to the ONCONFIG file do not take effect until you shut down and restart the database server, also called *reinitializing shared memory*. If you change a configuration parameter but do not reinitialize shared memory, the effective configuration differs from what the **onstat -c** option displays.

The values of the configuration parameters are stored in the file indicated by the **ONCONFIG** environment variable or, if you have not set the **ONCONFIG** environment variable, in **$INFORMIXDIR/etc/onconfig** on UNIX.

onutil CHECK RESERVED

The database server also stores current configuration information in the PAGE_CONFIG reserved page. The reserved page contains a description of the current, effective configuration.

To list the reserved page, execute **onutil** CHECK RESERVED.

Figure 3-1 shows sample output.

If you change the configuration parameters from the command line and run **onutil** CHECK RESERVED before you reinitialize shared memory, **onutil** discovers that values in the configuration file do not match the current values in the reserved pages and returns a warning message.

```
...
Validating Informix database server reserved pages - PAGE_CONFIG
    ROOTNAME                        rootdbs
    ROOTPATH                        /home/dyn_srv/root_chunk
    ROOTOFFSET                      0
    ROOTSIZE                        8000
    MIRROR                          0
    MIRRORPATH
    MIRROROFFSET                    0
    PHYSDBS                         rootdbs
    PHYSFILE                        1000
    LOGFILES                        5
    LOGSIZE                         500
    MSGPATH                         /home/dyn_srv/online.log
    CONSOLE                             /dev/ttyp5
...                                 ...
```

Figure 3-1
PAGE_CONFIG Reserved Page

Chapter

4

Configuration Parameters

In This Chapter

This chapter provides an overview of the ONCONFIG configuration parameters that the database server uses. The chapter can help you decide which parameters are most crucial for your particular environment and which parameters you can defer until you are tuning the performance of your database server. For details on each parameter, see the chapter on configuration parameters in the *Administrator's Reference*.

Chapter 5, "Configuring the Database Server," provides an overview of the configuration issues related to coservers.

Disk-Space Parameters

The disk-space parameters control how the database server manages storage space.

Root Dbspace

The first storage space that you allocate is called the *root database space*, or *root dbspace*. It stores all the basic information that describes your database server. Use the following parameters to describe the root dbspace.

Configuration Parameter	Description
ROOTNAME	Specifies the name of the root dbspace. You can choose any descriptive name for ROOTNAME, but it is usually called **rootdbs**. For more information, see "Root Dbspace" on page 15-17. The ROOTNAME parameter overrides the ROOTSLICE parameter for a particular coserver specified in the ONCONFIG file. You can choose any descriptive name for the ROOTNAME, but it must be unique for each database server instance.
ROOTOFFSET	Specifies an offset. For information about when to set ROOTOFFSET, refer to "Specifying an Offset" on page 16-7. When they are global parameters, each root dbspace has the same offset on disk for each coserver. You can override the global values within each coserver-specific section of the ONCONFIG file.

(1 of 2)

Configuration Parameter	Description
ROOTPATH	Specifies the pathname of the storage allocated to the root dbspace. For information on how to choose and allocate the storage, see "Allocating Disk Space" on page 16-6. The ROOTPATH is unique within a single coserver. The pathname can be the same for all coservers. The operating system keeps track of the actual disk that the node owns.
ROOTSIZE	Specifies the amount of space allocated to the root dbspace. For information on how to choose an appropriate size for the root dbspace, see "Size of the Root Dbspace" on page 15-36. When they are global parameters, each root dbspace is the same size for each coserver. You can override the global values within each coserver-specific section of the ONCONFIG file.
ROOTSLICE	Specifies a name for the root dbslice. The ROOTSLICE parameter is mutually exclusive with the ROOTNAME coserver-specific configuration parameter. You cannot specify ROOTSLICE in the coserver-specific section of the ONCONFIG file. For more information, see "Rootslices" on page 15-21.

(2 of 2)

Mirror of Root Dbspace

Mirroring allows fast recovery from a disk failure while the database server remains in on-line mode. When mirroring is active, the same data is stored on two disks simultaneously. If one disk fails, the data is still available on the other disk. Use the following parameters to describe mirroring of the root dbspace.

Configuration Parameter	Description
MIRROR	Defines whether mirroring is enabled or disabled.
MIRRORPATH	Specifies the full pathname of the chunk that mirrors the initial chunk of the root dbspace. You can specify MIRRORPATH in the coserver-specific section to override the global value and provide a different pathname for a specific coserver.
MIRROROFFSET	Specifies the offset into the device that serves as the mirror for the initial root dbspace chunk. For more information, see "Specifying an Offset" on page 16-7. You can also specify MIRROROFFSET in the coserver-specific section to override the global value if you want a different offset for a specific coserver.

Number of Storage Spaces

Use the following parameters to specify the maximum number of storage spaces and control the size of the safewrite area.

Configuration Parameter	Description
CONFIGSIZE	Specifies the size of the safewrite area, which is where the database server stores coserver-configuration information and other data that is required for maintaining data consistency across coservers. For more information, refer to the configuration chapter in the *Administrator's Reference*.
MAX_CHUNKS	Allows you to increase the maximum number of chunks up to 32767. For more information, see "Limiting Chunk Size and Number" on page 16-22.
MAX_DBSLICES	Allows you to increase the maximum number of dbslices up to 2047. For more information, see "Increasing the Number of Dbslices" on page 16-17.
MAX_DBSPACES	Allows you to increase the maximum number of dbspaces up to 32767. For more information, see "Specifying Names and Maximum Number of Storage Spaces" on page 16-14.

Other Space-Management Parameters

Use the following parameters to specify how the database server should manage particular types of disk space.

Configuration Parameter	Description
DBSPACETEMP	Specifies a list of dbspaces that the database server can use for the storage of temporary tables. For more information, see "Creating a Temporary Dbspace" on page 16-15.
FILLFACTOR	Specifies how much to fill index pages when indexes are created. For more information, see your *Performance Guide*.
ONDBSPACEDOWN	Defines how the database server treats a disabled dbspace that is not a critical dbspace.

Database Server Identification Parameters

Use the SERVERNUM and DBSERVERNAME parameters to provide unique identification for each instance of the database server.

Configuration Parameter	Description
DBSERVERALIASES	Specifies an alternate name or names for an instance of the database server. For information about using DBSERVERALIASES to create multiple listen endpoints to which clients can connect, see "Listen and Poll Threads for the Client / Server Connection" on page 11-28.
DBSERVERNAME	Specifies the unique name of an instance of the database server. Use the DBSERVERNAME for your **INFORMIX-SERVER** environment variable and in the **sqlhosts** information. On Extended Parallel Server, multiple coservers can exist within a single database server. The database server uniquely identifies each coserver by appending the coserver number to the DBSERVERNAME. Use DBSERVERNAME with the coserver number for client connections. For more information about client connections, see "Coserver Client Connections" on page 6-11.
SERVERNUM	Specifies a unique integer for the database server instance. The database server uses SERVERNUM to determine the shared-memory segment addresses within each coserver. Specify SERVERNUM only once for the entire instance of the database server (not for each coserver).

Warning: *Do not change the DBSERVERNAME configuration parameter without reinitializing the database server.*

Logging Parameters

Use the logging parameters to control the logical and physical logs.

Logical Log

The logical log contains a record of changes made to a database server instance. The logical-log records are used to roll back transactions, recover from system failures, and so on. The following parameters describe logical logging.

Configuration Parameter	Description
LOGBUFF	Determines the amount of shared memory reserved for the buffers that hold the logical-log records until they are flushed to disk. For information on how to tune the logical-log buffer, see “Logical-Log Buffer” on page 13-22.
LOGFILES	Specifies the number of logical-log files used to store logical-log records until they are backed up on disk.
LOGSIZE	Specifies the size of each logical-log file.
LOGSMAX	Specifies the maximum (not the actual) number of log files that you expect to have. For more information, see “Adding a Logical-Log File or Logslice” on page 21-4.
LTXHWM	Specifies the percentage of the available logical log that can be used before the database server takes moderate action to avoid the undesirable effects of reaching the point of LTXEHWM, the long-transaction exclusive-access high-water mark. For more information, see “Logical Log and Long Transactions” on page 20-15.
LTXEHWM	Specifies the point at which the database server takes drastic action.

For more information about these parameters, refer to “Size and Number of Logical-Log Files” on page 20-7.

Physical Log

The physical log contains images of all pages (units or storage) changed since the last checkpoint. The physical log combines with the logical log to allow fast recovery from a system failure. Use the following parameters to describe the physical log.

Configuration Parameter	Description
PHYSBUFF	Determines the amount of shared memory reserved for the buffers that serve as temporary storage space for pages about to be modified.
PHYSDBS	Specifies the name of the dbspace in which the physical log resides on each coserver. Any occurrence of the PHYSDBS parameter overrides the PHYSSLICE parameter for a particular coserver specified in the ONCONFIG file.
PHYSFILE	Specifies the size of the physical log. You can specify a coserver-specific PHYSFILE parameter if you want a different physical log size for a specific coserver.
PHYSSLICE	Specifies the name of the dbspace or dbslice on each coserver that contains the physical log. You cannot specify PHYSSLICE in the coserver-specific section of the ONCONFIG file.

For more information, see Chapter 22, “Physical Logging.”

Storage-Space and Logical-Log Backups

To create storage-space and logical-log backups of database server data, you can use the following tools. To verify storage-space backups, use ON-Bar.

Tool	Reference
ON-Bar	*Backup and Restore Guide*
onutil utility	*Administrator's Reference*

Message-Log Parameters

The message files provide information about how the database server is functioning.

Configuration Parameter	Description
CONSOLE	Specifies the pathname for console messages. For additional information, refer to "System Console" on page 2-13.
MSGPATH	Specifies the pathname of the database server message-log file. For more information, refer to "Message Log" on page 2-12.

Each coserver writes status and diagnostic messages to this file. Monitor the messages in this file regularly.

Informix recommends that you place the message-log file in a file system to which all coservers have access so that all messages go to the same file. When you centralize the message log, you monitor only one file for messages that any coserver generates.

Avoid placing message-log files in local file systems. Doing so complicates administration of the database server because it scatters log messages across multiple files and prevents you from viewing messages in sequence.

Shared-Memory Parameters

The shared-memory parameters affect database server performance.

Shared-Memory Size Allocation

Use the following parameters to control how and where the database server allocates shared memory.

Configuration Parameter	Description
SHMADD	Specifies the increment of memory that is added when the database server requests more memory
SHMBASE	Specifies the shared-memory base address and is computer dependent. Do not change its value.
SHMTOTAL	Specifies the maximum amount of memory that the database server is allowed to use.
SHMVIRTSIZE	Specifies the size of the first piece of memory that the database server attaches.

For more information on these parameters, see Chapter 13, "Shared Memory."

For platform-specific information on these database server shared-memory configuration parameters, refer to your machine notes file.

Shared-Memory Space Allocation

Use the following parameters to control how space is allocated in shared memory.

Configuration Parameter	Description
BUFFERS	Specifies the number of shared-memory buffers available to the database server. See "Shared-Memory Buffer Pool" on page 13-20.
CKPTINTVL	Specifies the maximum time interval allowed to elapse before a checkpoint.
DD_HASHMAX	Specifies the maximum number of entries for each hash bucket in the data-dictionary cache. For more information about setting DD_HASHMAX, refer to your *Performance Guide*.
DD_HASHSIZE	Specifies the number of hash buckets in the data-dictionary cache. For more information about setting DD_HASHSIZE, refer to your *Performance Guide*.
ISOLATION_LOCKS	Specifies the maximum number of rows that can be locked on a single scan when Cursor Stability isolation level is in effect. For performance considerations when you use this parameter, refer to your *Performance Guide*.
LBU_PRESERVE	Specifies the logs-full high-water mark. This feature preserves log space for administrative tasks. For a discussion of the logs-full high-water mark, refer to "Setting High-Water Marks" on page 20-18.
LOCKS	Specifies the initial number of locks available to database server user processes during transaction processing.
PAGESIZE	Specifies the database server page size.
PC_POOLSIZE	Specifies the number of SPL routines that can be stored in the SPL routine cache.

(1 of 2)

Configuration Parameter	Description
PC_HASHSIZE	Specifies the number of hash buckets in the SPL routine cache.
RESIDENT	Specifies whether shared-memory residency is enforced.
STACKSIZE	Specifies the stack size for database server user threads. For a discussion of the use of stacks, refer to "Stacks" on page 13-31.

(2 of 2)

Shared-Memory Buffer Control

Use the following parameters to control the shared-memory buffer pool.

Configuration Parameter	Description
LRUS, LRU_MAX_DIRTY, LRU_MIN_DIRTY	Describe the shared-memory pool of pages (memory spaces) that the database server uses. These parameters relate to LRU (least recently used) queues. See "LRU Queues" on page 13-36.
CLEANERS	Controls the number of threads used to flush pages to disk and return the pages to the shared-memory pool. See "Flushing Data to Disk" on page 13-45.
RA_PAGES and RA_THRESHOLD	Control the number of disk pages that the database server reads ahead during sequential scans. See "Configuring the Database Server to Read Ahead" on page 13-41.
IDX_RA_PAGES and IDX_RA_THRESHOLD	Control the number of index pages that the database server reads ahead during sequential scans. See "Configuring the Database Server to Read Ahead" on page 13-41.

Decision-Support Parameters

When you configure virtual shared memory on your system, you must decide what portion to reserve for decision-support queries. Decision-support queries use large amounts of the virtual portion of shared memory to perform joins and sort operations.

On Extended Parallel Server, parallel execution automatically occurs when the database operation involves data that is fragmented across multiple dbspaces and multiple CPU VPs are available.

Use the following parameters to control how decision-support queries are processed and to control the amount of memory that the database server allocates to decision-support queries. For more information about tuning these configuration parameters, refer to your *Performance Guide*.

Configuration Parameter	Description
DATASKIP	Controls whether the database server skips an unavailable table fragment.
DS_ADM_POLICY	Specifies how the Resource Grant Manager (RGM) should schedule queries.
DS_MAX_QUERIES	Specifies the maximum number of queries that can run concurrently.
DS_TOTAL_MEMORY	Specifies the amount of memory available for PDQ queries. Set the DS_TOTAL_MEMORY configuration parameter to any value not greater than the quantity (`SHMVIRTSIZE - 10 megabytes`).
MAX_PDQPRIORTY	Limits the amount of resources that a query can use.
OPTCOMPIND	Advises the optimizer on an appropriate join strategy for your applications.
PDQPRIORTY	Requests an amount of memory that a query can use.

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

For DSS-only applications that do not need to balance resources against OLTP applications, you can allocate all of the virtual portion to your decision-support queries. Set the DS_TOTAL_MEMORY configuration parameter to any value not greater than the quantity (`SHMVIRTSIZE - 10 megabytes`).

For strategies to improve performance with fragmentation and PDQ, refer to your *Performance Guide*.

Database Server Process Parameters

Configuration parameters for database server processes describe the type of processors on your computer and specify the behavior of virtual processes.

Processor Type

Use the following parameters to specify the type of processors in your environment and to allocate the virtual processors.

You need to set the following parameters to specific values, depending upon the number of processors on each node of your parallel-processing platform:

- MULTIPROCESSOR
- NUMAIOVPS
- NUMCPUVPS
- SINGLE_CPU_VP

For guidelines on setting these parameters, refer to "Setting Virtual-Processor Configuration Parameters with a Text Editor" on page 12-4.

Configuration Parameter	Description
MULTIPROCESSOR	Specifies the appropriate type of locking.
NETTYPE	Provides tuning options for each communications protocol.
NOAGE	Specifies whether priority aging should be in effect.
NUMAIOVPS	Specifies the number of virtual processors for AIO (asynchronous I/O) class threads.
NUMCPUVPS	Specifies the number of virtual processors for CPU class threads.
NUMFIFOVPS	Specifies the number of virtual processors of the FIFO class to run. For more information, refer to "First-In-First-Out Virtual Processor" on page 11-33.
SINGLE_CPU_VP	Specifies that the database server is using only one processor and allow the database server to optimize for that situation.

Processor Affinity

Some multiprocessor computers support *processor affinity*, which allows you to bind CPU virtual processors to CPUs. Use the following parameters to establish processor affinity.

Configuration Parameter	Description
AFF_NPROCS	On multiprocessor computers that support *processor affinity*, specifies the number of CPUs to which the database server can bind CPU virtual processors.
AFF_SPROC	On parallel-processing platforms that support *processor affinity*, specifies the CPU at which the coserver starts binding CPU virtual processors to CPUs.

Time Intervals

Use the following parameter to control the time intervals that the database server uses while processing transactions.

Configuration Parameter	Description
USEOSTIME	Controls the granularity of time reported by the database server.

Restore Parameters

Use the following parameters to control the number of threads that the database server allocates to off-line and on-line logical recovery.

Configuration Parameter	Description
OFF_RECOVERY_THREADS	Specifies the number of recovery threads used during a cold restore.
ON_RECOVERY_THREADS	Specifies the number of recovery records used during fast recovery and a warm restore.

Event-Alarm Parameters

The database server can execute a program if a noteworthy event occurs. Noteworthy events include failure of database, table, index, simple large object, or smart large object; chunk or dbspace taken off-line; internal subsystem failure; initialization failure; and detection of long transaction.

Use the following parameter to specify what actions to take when noteworthy events occur.

Configuration Parameter	**Description**
ALARMPROGRAM	Specifies the location of a file that is executed when an event alarm occurs.

Dump Parameters

Use the following parameters to control the types and location of core dumps that are performed if the database server fails.

Configuration Parameter	**Description**
DUMPCNT	Specifies the number of assertion failures for which a single thread dumps shared memory.
DUMPCORE	Controls whether assertion failures cause a virtual processor to dump core memory.
DUMPDIR	Specifies a directory where the database server places dumps of shared memory, **gcore** files, or messages from a failed assertion.
DUMPGCORE	If your operating system supports the **gcore** utility, an assertion failure causes the database server to call **gcore**.
DUMPSHMEM	Specifies that shared memory should be dumped on an assertion failure.

Coserver Parameters

Use the following parameters to set up coservers.

Configuration Parameter	Description
COSERVER	Specifies the numeric identification of a coserver. The coserver number must be unique among all coservers within the database server. Assign coserver numbers consecutively, starting with 1.
END	Specifies the end of each coserver-specific, BAR_SM specific, or storage-manager specific section of the ONCONFIG file.
NODE	Specifies the host name of the node on which the coserver runs. Assign each coserver to a node. The host name is located in the operating system hosts file. For more information about the hosts file, refer to "Connectivity Files" on page 6-13.

Specialized Parameters

Some parameters appear in the configuration file only when you use specialized features of the database server.

Optical Media

Informix Storage Manager allows you to back up information to optical media, but it does not allow the database server to access directly the data that is stored on the disks.

UNIX

Some UNIX platforms have additional configuration parameters. For a description of these specialized parameters and instructions for using them, see your machine notes.

Chapter

Configuring the Database Server

5

In This Chapter

This chapter discusses the preparation tasks that are unique to Extended Parallel Server.

This chapter describes the following:

- how the ONCONFIG configuration file for multiple coservers differs from the default configuration file.
- the various ways that you can configure coservers and the considerations that are involved in choosing one sort of coserver configuration over another.

Configuring Multiple Coservers

For a multiple-coserver configuration, the database server contains one centralized configuration file that contains two sections of configuration parameters:

- Global configuration parameters

  After you specify these parameters once, they apply equally to all coservers.
- Coserver-specific configuration parameters

  The coserver-specific section includes parameters that apply to individual coservers. If you intend to configure your database server with multiple coservers, you must include a coserver-specific section in your ONCONFIG file.

If a configuration parameter in the coserver-specific section conflicts with a parameter specified in the global section, the coserver-specific value overrides the global value only for that particular coserver.

Global Configuration Parameters

You specify the global configuration parameters only in the global section to apply to all of the coservers in your database server system. The database server uses the parameter value in the global section as the default value for the parameter on every coserver. For more information on specific parameters, see Chapter 4, "Configuration Parameters."

For the initial configuration of the database server, you can leave most of the global parameters set to their default values. You *must* review and change, if necessary, parameters for the following items:

- Root and log dbspaces on multiple coservers
- Database server identification
- Processors
- Message files
- Shared-memory size allocation
- ON-Bar backup and restore

You can specify the following configuration parameters in both the global and coserver-specific section:

- AFF_SPROC
- AFF_NPROCS
- MIRROROFFSET
- MIRRORPATH
- PHYSDBS
- PHYSFILE
- ROOTNAME
- ROOTOFFSET
- ROOTPATH
- ROOTSIZE

Tip: *Informix recommends that you use the global configuration parameters ROOTSLICE and PHYSSLICE rather than coserver-specific ROOTNAME and PHYSDBS values because the global parameters are easier to define and manage. Override the values of global parameters coserver-specific section only if necessary.*

Root and Log Dbspaces on Multiple Coservers

Extended Parallel Server uses the ROOTSLICE and PHYSSLICE configuration parameters to configure root and physical log dbspaces across a large number of coservers. Specify these parameters only once in the global section of the configuration file rather than once for each coserver that you define.

Using Formatting Characters with ROOTPATH and MIRRORPATH

You can use embedded formatting characters in the ROOTPATH and MIRRORPATH configuration parameters to generate uniform chunk names for root and log dbspaces across a large number of coservers.

The pathname specification with the embedded formatting characters is referred to as a *pathname-format*. You can embed the following formatting characters in a pathname-format:

`%c`	is a formatting string that is replaced with the coserver-number of the coserver on which a dbspace is to be created. The coserver-number is the value that you specified in the COSERVER configuration parameter for this coserver. If the number is less than 10, the number does not have a leading zero because it is handled as a character rather than an integer.
`%n`	is a formatting string that is replaced with the host name of the node for the coserver on which a dbspace is to be created. The host name is the value that you specified in the NODE configuration parameter for this coserver.

Coserver-Specific Configuration Parameters

The parameters that describe each coserver are at the bottom of the ONCONFIG configuration file. When you specify the ROOTSLICE and PHYSSLICE global configuration parameters, you usually need to specify only the following coserver-specific parameters:

- COSERVER
- NODE
- END

Figure 5-1 shows an excerpt of the coserver-specific section in a sample ONCONFIG file.

```
# Global Parameters
ROOTSLICE rootdbs
...
PHYSSLICE rootdbs
...
# Coserver-specific parameters
COSERVER        1
NODE            octopus1
END

COSERVER        2
NODE            octopus2
END
...

COSERVER        8
NODE            eagle8
END
```

Figure 5-1
Coserver-Specific Section of an ONCONFIG File

Important: *The ONCONFIG file for a single-coserver configuration on Extended Parallel Server contains no coserver-specific section.*

Platform-Specific Configuration Parameters

Additional coserver-specific parameters might be listed in the machine notes file for certain platforms.

If your communication interface between nodes requires configurable buffers, you also need to consider the amount of space that these message buffers take up in the virtual portion on each coserver. Additional configuration parameters (such as HADDR, SADDR, and LADDR) specify sizes of these message buffers and apply only to certain computer platforms. You specify these buffer parameters in the coserver-specific section.

Some configuration parameters (such as ASYNCRQT, SENDEPDS, and DGINFO) apply only to certain computer platforms. ASYNCRQT and SENDEPDS are global parameters; ASYNCRQT specifies the number of outstanding receive requests and SENDEPDS the number of send endpoints per CPU VP. DGINFO passes additional optional, platform-specific information to the Datagram layer.

To determine if your computer requires these additional parameters, consult your machine notes file.

Organizing the Configuration File

Figure 5-2 shows an excerpt from the global, coserver-specific, and storage-manager specific sections in a sample ONCONFIG file. This sample configuration uses eight nodes with a single coserver on each node.

Figure 5-2
Global and Coserver-Specific Sections of an ONCONFIG File

```
DBSERVERNAME    eds

ROOTSLICE       rootdbs
ROOTPATH        /work/dbspaces/rootdbs_%c
ROOTOFFSET      0
ROOTSIZE        40000

MIRROR          1       # 1 = yes
MIRRORPATH      /work/dbspaces/mirror_%c
MIRROROFFSET    0

PHYSSLICE       rootdbs
PHYSFILE        8000

LOGFILES        3
LOGSIZE         1000
...

COSERVER        1
    NODE        octopus1
    ...

END
...

COSERVER        8
    NODE        eagle8
    ...

END

# The storage manager can access onbar-worker processes on
# coservers 1 through 8.
BAR_SM              1
  BAR_WORKER_COSVR  1-8
END
```

The configuration parameters that Figure 5-2 shows generate the following dbspace names and pathnames for the root dbspaces:

```
coserver 1  rootdbs.1   /work/dbspaces/rootdbs_1
                        /work/dbspaces/mirror_1
...
coserver 8  rootdbs.8   /work/dbspaces/rootdbs_8
                        /work/dbspaces/mirror_8
```

Backups go to the storage manager on **coserver 1**.

Setting Storage-Manager Parameters for ON-Bar

Extended Parallel Server allows you to define multiple storage-manager instances, but only one instance per node. You can configure and use different storage-manager brands for different purposes. (ON-Bar works with a storage manager to back up and restore data.)

The parameters that describe each storage manager are in the storage-manager section of the ONCONFIG file. Each storage-manager section begins with the BAR_SM parameter and ends with the END parameter. The BAR_SM parameter cannot be embedded in the coserver-specific section or nested.

If you define one storage manager on Extended Parallel Server, specify the storage-manager number in the BAR_SM parameter and specify the coservers where you want to run ON-Bar processes in the BAR_WORKER_COSVR parameter.

If you define multiple storage managers, you need to set the storage-manager parameters for each node.

However, you can specify certain storage-manager parameters in the global section of the ONCONFIG file if you want to use the same values for all storage-manager instances.

The following table shows the storage-manager specific parameters for ON-Bar. For more information, refer to the *Backup and Restore Guide*.

Configuration Parameter	Description	Can be storage-manager specific	Always storage-manager specific
BAR_DBS_COSVR	Specifies coservers that send backup and restore data to the storage manager.		✔
BAR_IDLE_TIMEOUT	Specifies the maximum number of minutes that an **onbar-worker** process is idle before it is shut down.	✔	
BAR_LOG_COSVR	Specifies coservers that send log backup data to the storage manager.		✔
BAR_SM	Specifies the storage-manager number.		✔
BAR_SM_NAME	Specifies the storage-manager name.		✔
BAR_WORKER_COSVR	Lists the coservers that can access the storage manager.		✔
BAR_WORKER_MAX	Specifies the maximum number of **onbar-worker** processes started for a storage manager.	✔	
LOG_BACKUP_MODE	Specifies whether to use manual or continuous logical-log backups or to turn off logical-log backups. Set this parameter in the ONCONFIG file.	✔	

Figure 5-3 shows an excerpt of the storage-manager section in a sample ONCONFIG file for two storage managers, ABEL and BAKER. This configuration starts ON-Bar processes on **coservers 1**, **2**, and **3**. **Coservers 1** and **2** are on the same node. **Coserver 3** is on a different node. You can optionally override global configuration parameters with the storage-manager specific configuration parameters.

```
# storage manager ABEL
BAR_SM   1
BAR_WORKER_COSVR 1,2
END

# storage manager BAKER
BAR_SM   2
BAR_WORKER_COSVR 3
END
```

Figure 5-3
Storage-Manager Specific Section of an ONCONFIG File

Choosing a Coserver Configuration

This book uses the term *platform* to indicate the combination of the operating system and its underlying hardware.

Depending on the architecture of your computer or platform, you can choose among the following options for configuring coservers:

- A single coserver on a single-node platform
- Multiple coservers on a single-node platform
- Single coservers on each node of a multiple-node platform
- Multiple coservers on each node of a multiple-node platform

Each of these options is best suited to a particular platform architecture, as the following sections describe. For configuration purposes, a *node* is a computer system that contains one or more processors and has a separate network address, separate random-access memory, and access to owned disks for storage. That computer system could be a uniprocessor, an SMP computer, an independent subsystem within an SMP computer, or a single computer within an MPP platform.

Single Coserver on a Single-Node Platform

A configuration that consists of a single coserver on a single node is best suited to a uniprocessor or an SMP computer that:

- has approximately 10 or fewer CPUs (in the case of an SMP computer).
- has an amount of physical memory that is comparable in size to the address space of a single process.
- has enough local disk space to support the data that is required for the intended application.

If your configuration consists of multiple small nodes, configure one coserver per node. A single coserver on a single node is easy to administer. You do not need a coserver-specific section in the ONCONFIG file. You can use the **onconfig.std** file as a template for a single coserver configuration.

Parallel processing can occur within a single coserver. The NUMCPUVPS parameter indicates the number of CPUs that are available to the database server for parallel processing.

Multiple Coservers on a Single-Node Platform

A configuration of multiple coservers on a single-node platform is best suited to an SMP computer that:

- has over 8 CPUs or an amount of physical memory that is at least double the size of the address space of a single process (4 gigabytes for 32-bit operating systems or 8 gigabytes for 64-bit operating systems).
- cannot be partitioned into subsystems to create separate nodes.
- has enough local disk space to support the data that is required for the intended application.

If your configuration consists of a few nodes with lots of CPUs, memory, and disk space, configure multiple coservers per node. Each coserver requires a coserver section in the ONCONFIG file. You can use the **onconfig.xps** file as a template for a multiple coserver configuration.

Ensure that each coserver can use the disks independently for parallel I/O. Also ensure correct access permissions for dbspaces and files on each coserver. The use of multiple coservers can help to balance the processing load across the available memory and CPU resources. Multiple coservers are also useful for logging-intensive applications or situations in which extremely rapid recovery from any failure is an urgent requirement.

Parallel processing can occur both within coservers and between coservers. To calculate the number of CPUs available to the database server, multiply the NUMCPUVPS parameter by the number of COSERVER parameters listed in the ONCONFIG file.

To run multiple coservers on a single node

1. Install one copy of the database server on a computer (**icecream9** in this example).
2. To define the new coservers, add a coserver-specific section to your ONCONFIG file, as the following example shows.

   ```
   COSERVER  1,2,3
       NODE   icecream9
       ...
       END
   ```
3. Add an entry to your **sqlhosts** file to enable the new coserver to accept client connections.

   For the **hostname** field of the **sqlhosts** file, use the same value that you specified in the NODE configuration parameter in your ONCONFIG file.

   For the **dbservername** field, use the coserver name, which is composed of the value of the DBSERVERNAME configuration parameter followed by a period and the number of the new coserver.

   ```
   dbservername.coserver-number
   ```

   For more information, see "The sqlhosts File" on page 6-21.
4. Initialize the database server with the following command:

   ```
   xctl -C oninit -iy
   ```

Single Coservers on Each Node of a Multiple-Node Platform

This configuration is best suited to parallel-processing platforms composed of nodes that:

- each have 10 or fewer CPUs.
- each have an amount of physical memory that is comparable in size to the address space of a single process.
- each have enough local disk space to support an equal portion of the data that is required for the intended application, plus a suitable amount of space for temporary tables and sort files.

Important: *For ease of administration, Informix recommends that you use the same hardware configuration for all Extended Parallel Server nodes.*

The use of multiple nodes allows the workload to be distributed across a high-speed interconnect, network, or bus (in the case of a partitioned SMP computer). Communication is not limited to client connections.

Each coserver requires a coserver-specific section in the ONCONFIG file. Each node requires an entry in the **sqlhosts** connectivity file and an entry in the operating system **hosts** network file. For more information, refer to "Network-Configuration Files" on page 6-13.

Multiple coservers allow parallel processing between the nodes to provide parallelism across coservers. To calculate the number of CPUs available to the database server, you can multiply NUMCPUVPS by the number of COSERVER parameters listed in the ONCONFIG file.

Multiple Coservers on Each Node of a Multiple-Node Platform

A configuration of multiple coservers on each node is best suited to parallel-processing platforms composed of nodes that:

- each have over 8 CPUs or an amount of physical memory that is at least double the size of the address space of a single process.
- cannot be partitioned to form separate nodes.
- each have enough local disk space to support an equal portion of the data that is required for the intended application, plus a suitable amount of space for temporary tables and sort files.

The use of multiple coservers on each node can help to balance the processing load across the available memory and CPU resources. Each coserver requires a coserver section in the ONCONFIG file. Each node requires an entry in the **sqlhosts** connectivity file and an entry in the operating system **hosts** network file. For more information, refer to "Network-Configuration Files" on page 6-13.

Parallel processing can occur both within coservers and between coservers. To calculate the number of CPUs available to the database server, you can multiply NUMCPUVPS by the number of COSERVER parameters listed in the ONCONFIG file.

Adding Coservers

You might want to add coservers to the database server in the following situations:

- When you increase the number of nodes on your system and add them to the database server
- When you reassign existing nodes to your database server
- When you want to add another coserver to a node

  A node can contain more than one coserver.

To add a coserver

1. Perform a level-0 backup of the database server before you add a coserver. For more information, see the *Backup and Restore Guide*.
2. Unload the tables into external tables using Informix internal format. Use statements similar to the following ones:

   ```
   SELECT * FROM employee
   INTO EXTERNAL emp_txt
   USING (
      FORMAT 'INFORMIX',
      DATAFILES ("DISK:cogroup_all:/work2/mydir/emp.dat")
      );
   ```

   For more information, see the chapter on loading with external tables in the *Administrator's Reference*.
3. Use the **dbschema** utility to capture the database schema. For information on using **dbschema**, see the *Informix Migration Guide*.

   ```
   dbschema -d <database_name>
   ```
4. Bring the database server off-line.

   ```
   xctl onmode -ky
   ```

   For more information on the **xctl** command-line utility, see the utilities chapter in the *Administrator's Reference*.
5. Edit your ONCONFIG file to define the new coserver. Add a coserver-specific section to your ONCONFIG file for the new coserver as the following example shows.

   ```
   COSERVER   9
       NODE   icecream9
       ...
       END
   ```

   For more information on the COSERVER and NODE configuration parameters, see "Coserver-Specific Configuration Parameters" on page 5-5.

6. Add an entry to your **sqlhosts** file to enable the new coserver to accept client connections.

   For the **hostname** field of the **sqlhosts** file, use the same value that you specified in the NODE configuration parameter in your ONCONFIG file.

   For the **dbservername** field, use the coserver name, which is composed of the value of the DBSERVERNAME configuration parameter followed by a period and the number of the new coserver.

   ```
   dbservername.coserver-number
   ```

   For more information on the **sqlhosts** file, see "The sqlhosts File" on page 6-21.

7. Reinitialize the database server using the following command:

   ```
   xctl -C oninit -iy
   ```

8. Use the ALTER DBSLICE command to extend the dbslice to the new coserver.

9. Optionally, edit the schema file. Use the TABLE FRAGMENT statement to redefine the table fragmentation so that the database server automatically stores the rows from the external table across all the dbspaces in the dbslice.

10. Use the dbschema file to recreate the database schema.

11. Alter the tables to type RAW.

    ```
    CREATE RAW TABLE emp_raw ...
    ```

12. Use the CREATE EXTERNAL TABLE statement to create the external tables.

13. Use express mode to load the external tables into the database as the following example shows.

    ```
    INSERT INTO employee SELECT * FROM EXTERNAL TABLE
    emp_txt;
    ```

14. If you do not plan to update the tables, change them to type STATIC. If you plan to update the tables, change them to a logging table such as OPERATIONAL or STANDARD.

15. If you did not modify the database schema file, use the ALTER FRAGMENT command to refragment the tables and redistribute the data across the old and new coservers.

16. Perform a level-0 backup of the database server.

Defining Cogroups

Define cogroups to facilitate the management of multiple coservers in Extended Parallel Server. A cogroup is a set of coservers within the database server. You assign coservers to cogroups in order to manage them as a unit. Typically, you assign coservers that share tables to the same cogroup. A coserver can be a member of multiple cogroups.

Extended Parallel Server provides a system-defined cogroup that is called **cogroup_all**. The **cogroup_all** cogroup includes all the coservers that the ONCONFIG configuration file defines.

You might want to define a cogroup that consists of a subset of the coservers for a specific application or database.

To define a cogroup

1. Determine which coservers should be part of the cogroup.
2. Obtain the coserver names from your ONCONFIG configuration file. The coserver name has the following format:

   dbservername.coserver-number

   For *dbservername*, use the value of the DBSERVERNAME configuration parameter.

   For *coserver-number*, use the value of the COSERVER configuration parameter.

   For example, your ONCONFIG file defines 16 coservers as follows:

   ```
   DBSERVERNAME        eds
   ...
   COSERVER            1
       NODE            octopus1
       ...
       END
   ...
   COSERVER            16
       NODE            bear16
       ...
   END
   ```

   The coserver names are **eds.1**, **eds.2**, and so forth. For more information see "Coserver-Specific Configuration Parameters" on page 5-5.

3. Use the **onutil** command-line utility to create the cogroup.

   The following example shows **onutil** running with interactive input. The input is the **onutil** CREATE COGROUP command to create a cogroup that includes the first four coservers in the preceding sample ONCONFIG configuration file.

   ```
   %onutil
   1> CREATE COGROUP acctg_group
   2> from eds.1, eds.2, eds.3, eds.4;
   Cogroup successfully created.
   3> QUIT;
   ```

   For more information about the **onutil** CREATE COGROUP command, see the utilities chapter in the *Administrator's Reference*.

Modifying Cogroups

You might want to modify a cogroup when you want to change the coservers that it includes. This situation occurs when you add nodes to your platform.

To change the coservers in a cogroup

1. Use the **onutil** command-line utility to drop the cogroup.

   The following example shows the **onutil** DROP COGROUP command to drop an accounting cogroup:

   ```
   %onutil
   1>DROP COGROUP acctg_group;
   Cogroup successfully dropped.
   2> QUIT;
   ```

2. Edit your ONCONFIG file to define the new coservers.

   Add a coserver-specific section to your ONCONFIG file for each new coserver. For example, the following excerpt from the ONCONFIG file shows sample values that you might use to define new coservers:

   ```
   COSERVER        9
       NODE      bear9
       ...
   END
   ...
   COSERVER        16
       NODE      bear16
       ...
   END
   ```

   For more information about the COSERVER and NODE configuration parameters, see the *Administrator's Reference*.

3. Use the **onutil** command-line utility to create the modified cogroup.

   The following example shows **onutil** running with interactive input. The input is the **onutil** CREATE COGROUP command to create a cogroup that includes the first 12 coservers in the preceding sample ONCONFIG file.

   ```
   %onutil
   1> CREATE COGROUP acctg FROM eds.%r(1..12);
   Cogroup successfully created.
   2> QUIT;
   ```

   This example substitutes the coserver number for the `%r` formatting character in the coserver name in the FROM clause.

   This example assumes that you defined your dbservername as **eds** in your ONCONFIG file. Therefore, this example creates a cogroup that includes the following coservers:

   ```
   eds.1
   eds.2
   eds.3
   eds.4
   eds.5
   eds.6
   eds.7
   eds.8
   eds.9
   eds.10
   eds.11
   eds.12
   ```

   For more information about coserver names, the **onutil** CREATE COGROUP command, and the use of formatting characters, see the utilities chapter in the *Administrator's Reference*.

Monitoring Coserver Activities

Monitoring coserver activities includes:

- monitoring resources across all coservers.
- investigating resources on an individual coserver.

Monitoring resources across all coservers includes the following tasks:

- Verifying that the coservers are in online mode
- Monitoring the query execution across coservers
- Checking the status of the root dbspace and chunks across all coservers

Creating and Loading Tables Fragmented Across Coservers

After you have created dbslices and dbspaces, you can create tables fragmented across multiple coservers. For VLDBs that are fragmented across many coservers, system-defined hash on the join column is often a good first choice for a fragmentation scheme.

For more details on fragmentation strategies, refer to your *Performance Guide*. For details on how to use the CREATE TABLE statement to fragment tables, refer to the *Informix Guide to SQL: Syntax*.

Extended Parallel Server provides external tables that allow you to load multiple table fragments in parallel across multiple coservers. For more details on how to use these tables, refer to the chapter on loading with external tables in the *Administrator's Reference*.

Chapter

6 Client/Server Communications

In This Chapter

This chapter explains the concepts and terms that you need to understand in order to configure client/server communications. The chapter consists of the following parts:

- Description of client/server architecture
- Database server connection types
- Communication services
- Connectivity files
- ONCONFIG connectivity parameters
- Connectivity environment variables
- Examples of client/server configurations

Client/Server Architecture

Informix products conform to a software design model called *client/server*. The client/server model allows you to place an application or *client* on one computer and the database *server* on another computer, but they can also reside on the same computer. Client applications issue requests for services and data from the database server. The database server responds by providing the services and data that the client requested.

You use a *network protocol* together with a *network programming interface* to *connect* and transfer data between the client and the database server. The following sections define these terms in detail.

Network Protocol

A network protocol is a set of rules that govern how data is transferred between applications and, in this context, between a client and a database server. These rules specify, among other things, what format data takes when it is sent across the network. An example of a network protocol is TCP/IP.

The rules of a protocol are implemented in a *network-protocol driver.* A network-protocol driver contains the code that formats the data when it is sent from client to database server and from database server to client.

Clients and database servers gain access to a network driver by way of a *network programming interface.* A network programming interface contains system calls or library routines that provide access to network-communications facilities. An example of a network programming interface for UNIX is TLI (Transport Layer Interface). The power of a network protocol lies in its ability to enable client/server communication even though the client and database server reside on different computers with different architectures and operating systems.

You can configure the database server to support more than one protocol, but consider this option only if some clients use TCP/IP and some use IPX/SPX.

To determine the supported protocols for your operating system, see "Database Server Connection" on page 6-5.

To specify which protocol the database server uses, set the **nettype** field in the **sqlhosts** file.

Network Programming Interface

A network programming interface is an application programming interface (API) that contains a set of communications routines or system calls. An application can call these routines to communicate with another application that resides on the same or on different computers. In the context of this discussion, the client and the database server are the applications that call the routines in the TLI or sockets application-programming interface. Clients and database servers both use network programming interfaces to send and receive the data according to a communications protocol.

Both client and database server environments must be configured with the same protocol if client/server communication is to succeed. However, some network protocols can be accessed through more than one network programming interface. For example, TCP/IP can be accessed through either TLI or sockets, depending on which programming interface is available on the operating-system platform. Therefore, a client using TCP/IP through TLI on one computer can communicate with a database server using TCP/IP with sockets on another computer, or vice versa. For an example, see "Using a Network Connection" on page 6-48.

Database Server Connection

A *connection* is a logical association between two applications; in this context, between a client application and a database server. A connection must be established between client and database server *before* data transfer can take place. In addition, the connection must be maintained for the duration of the transfer of data.

Tip: *The Informix internal communications facility is called Association Services Facility (ASF). If you see an error message that refers to ASF, you have a problem with your connections.*

A client application establishes a connection to a database server with either the CONNECT or DATABASE SQL statement. For example, to connect to the database server **my_server**, an application might contain the following form of the CONNECT statement:

```
CONNECT TO '@myserver'
```

For more information on the CONNECT and DATABASE statements, see the *Informix Guide to SQL: Syntax*.

Multiplexed Connection

Some applications connect multiple times to the same database server on behalf of one user. A *multiplexed connection* uses a single *network* connection between the database server and a client to handle multiple *database* connections from the client. Client applications can establish multiple connections to a database server to access more than one database on behalf of a single user. If the connections are not multiplexed, each database connection establishes a separate network connection to the database server. Each additional network connection consumes additional computer memory and CPU time, even for connections that are not active. Multiplexed connections enable the database server to create multiple database connections without consuming the additional computer resources that are required for additional network connections.

To configure the database server to support mulitplexed connections, you must include in the ONCONFIG file a special NETTYPE parameter that has a value of `SQLMUX`, as in the following example:

```
NETTYPE SQLMUX
```

To configure the automatic use of multiplexed connections by clients, the entry in the **sqlhosts** file that the client uses for the database server connection must specify the value of `m=1` in the **options** field, as in the following example:

```
menlo ontlitcp valley jfk1 m=1
```

You do not need to make any changes to the **sqlhosts** file that the database server uses. The client program does not need to make any special SQL calls to enable connections multiplexing. Connection multiplexing is enabled automatically when the ONCONFIG file and the **sqlhosts** file are configured appropriately. For information on the NETTYPE configuration parameter, refer to the chapter on configuration parameters in the *Administrator's Reference*. For more information on the **sqlhosts** file, refer to "The sqlhosts File" on page 6-20.

The following limitations apply to multiplexed connections:

- Multithreaded client connections are not supported.
- Shared-memory connections are not supported.

- The ESQL/C **sqlbreak()** function is not supported.
- You can activate database server support for multiplexed connections only when the database server starts.

If any of these conditions exist when an application attempts to establish a connection, the database server establishes a standard connection. The database server does not return an SQL error.

Connections That the Database Server Supports

The database server supports the following types of connections to communicate between client applications and a database server

Connection Type	Local	Network
Sockets	✔	✔
TLI (TCP/IP)	✔	✔
TLI (IPX/SPX)	✔	✔
Shared memory	✔	
Stream pipe	✔	

On many UNIX platforms, the database server supports multiple network programming interfaces. To check which interface/protocol combinations the database server supports for your operating system, check the machine notes. The section "Machine Specific Notes" describes interface/protocol combinations that are available on your platform, similar to the following example:

```
Machine Specific Notes:
=======================

1. The following interface/protocol combinations(s) are
supported for
this platform:

             Berkeley sockets using TCP/IP
```

To set up a client connection

1. Specify connectivity configuration parameters in your ONCONFIG file.
2. Set up appropriate entries in the connectivity files on your platform.
3. Specify connectivity environment variables in your UNIX initialization scripts.
4. Define a dbserver group for your database server in the **sqlhosts** file.

The following sections describe database server connection types in more detail. For detailed information about implementing the connections described in the following sections, refer to the following topics:

- "Connectivity Files" on page 6-13
- "The sqlhosts Information" on page 6-22
- "ONCONFIG Parameters for Connectivity" on page 6-42
- "Environment Variables for Network Connections" on page 6-45

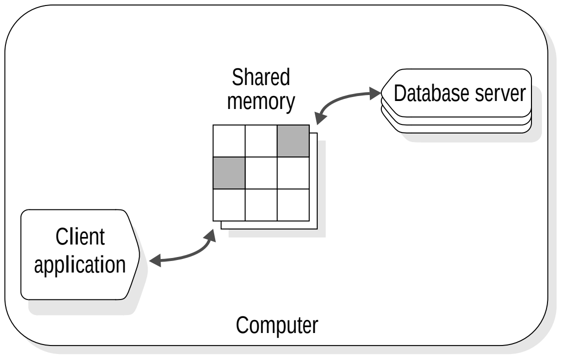
Local Connections

A *local connection* is a connection between a client and the database server on the same computer. The following sections describe these types of local connections.

Shared-Memory Connections

A *shared-memory connection* uses an area of shared-memory as the *channel* through which the client and database server communicate with each other. Figure 6-1 illustrates a shared-memory connection.

Figure 6-1
A Shared-Memory Connection



Shared memory provides fast access to a database server, but it poses some security risks. Errant or malicious applications could destroy or view message buffers of their own or of other local users. Shared-memory communication is also vulnerable to programming errors if the client application performs explicit memory addressing or overindexes data arrays. Such errors do not affect the database server if you use network communication or stream pipes. For an example of a shared-memory connection, refer to “Using a Shared-Memory Connection” on page 6-46.

A client cannot have more than one shared-memory connection to a database server.

For information about the portion of shared memory that the database server uses for client / server communications, refer to “Communications Portion of Shared Memory” on page 13-33. For additional information, you can also refer to “How a Client Attaches to the Communications Portion” on page 13-12.

Stream-Pipe Connections

A *stream pipe* is a UNIX interprocess communication (IPC) facility that allows processes on the same computer to communicate with each other. You can use stream-pipe connections any time that the client and the database server are on the same computer. For more information, refer to "Network Protocol" on page 6-25 and "Shared-Memory and Stream-Pipe Communication" on page 6-27.

Stream-pipe connections, unlike shared-memory connections, do not pose the security risk of being overwritten or read by other programs that explicitly access the same portion of shared memory.

Disadvantages of Stream-Pipe Connections

Stream-pipe connections have the following disadvantages:

- Stream-pipe connections might be slower than shared-memory connections on some computers.
- Stream pipes are not available on all platforms.

Local-Loopback Connections

A network connection between a client application and a database server on the same computer is called a *local-loopback* connection. The networking facilities used are the same as if the client application and the database server were on different computers. You can make a local-loopback connection provided your computer is equipped to process network transactions. Local-loopback connections are not as fast as shared-memory connections, but they do not pose the security risks of shared memory.

In a local-loopback connection, data appears to pass from the client application, out to the network, and then back in again to the database server. In fact, although the database server uses the network programming interface (TLI or sockets), the internal connection processes send the information directly between the client and the database server and do *not* put the information out on the network.

For an example of a local-loopback connection, see "Using a Local-Loopback Connection" on page 6-47.

Coserver Client Connections

All coservers in Extended Parallel Server can accept client connections. A coserver that accepts the connection for a particular client is called the *connection coserver* for that client.

To identify each connection coserver uniquely, the database server uses dbservernames of the following form:

```
dbservername.coserver_number
```

dbservername	is the value that you specify in the DBSERVERNAME or DBSERVERALIASES configuration parameter.
coserver_number	is the integer that you specify in each COSERVER configuration parameter.

This form of the dbservername is referred to as a *coserver name*. Extended Parallel Server uses the DBSERVERNAME and coserver numbers specified in the ONCONFIG file to generate coserver names. For example, if the DBSERVERNAME is **myxps** and the ONCONFIG file specifies eight coservers, the database server automatically generates the following coserver names:

```
myxps.1
...
myxps.8
```

Communication Support Services

Communication support services include connectivity-related services such as the following security services:

- Authentication is the process of verifying the identity of a user or an application. The most common form of authentication is to require the user to enter a name and password to obtain access to a computer or an application.
- Message integrity ensures that communication messages are intact and unaltered when they arrive at their destination.
- Message confidentiality protects messages, usually by encryption and decryption, from viewing by unauthorized users during transmission.

Communication support services can also include other processing such as data compression or traffic-based accounting.

The database server provides a default method of authentication, which is described in "Network-Security Files" on page 6-16. The database server uses the default authentication policy when you do not specify a communications support module.

The database server provides extra security-related communication support services through plug-in software modules called Communication Support Modules (CSM).

Informix Password Communication Support Module

The Informix Password Communication Support Module (PSWDCSM) provides encryption to protect a password when it must be sent between the client and the database server for authentication. PSWDCSM is available on all platforms.

To use password encryption, you must add a line to the CSS/CSM configuration file, **concsm.cfg**, and add an entry to the **options** column of the **sqlhosts** file. The **concsm.cfg** file must contain an entry for each communications support module that you are using. For information on the **concsm.cfg** file, see "CSM Configuration File" on page 6-19. For information on specifying the CSM in the **sqlhosts** file, see "Communication Support Module Option" on page 6-32.

Connectivity Files

The *connectivity files* contain the information that enables client/server communication. These files also enable a database server to communicate with another database server. The connectivity configuration files can be divided into three groups:

- Network-configuration files
- Network-security files
- The **sqlhosts** file

Network-Configuration Files

This section identifies and explains the use of network-configuration files on TCP/IP and IPX/SPX networks.

TCP/IP Connectivity Files

When you configure the database server to use the TCP/IP network protocol, you use information from the network-configuration files **hosts** and **services** to prepare the **sqlhosts** information.

The network administrator maintains these files. When you add a host, or a software service such as a database server, you need to inform the network administrator so that person can make sure the information in these files is accurate.

The **hosts** file needs a single entry for each network-controller card that connects a computer running an Informix client / server product on the network. Each line in the file contains the following information:

- Internet address (or ethernet card IP address)
- Host name
- Host aliases (optional)

Although the length of the host name is not limited in the **hosts** file, Informix limits the host name to 256 characters. Figure 6-7 on page 6-38 includes a sample **hosts** file.

The **services** file contains an entry for each service available through TCP / IP. Each entry is a single line that contains the following information:

- Service name

  Informix products use this name to determine the port number and protocol for making client / server connections. The service name can have up to 256 characters.
- Port number and protocol

  The port number is the computer port, and the protocol for TCP / IP is `tcp`.
- Aliases (optional)

The service name and port number are arbitrary. However, they must be unique within the context of the file and must be identical on all computers that are running Informix client / server products. The **aliases** field is optional. For example, a **services** file might include the following entry for a database server:

```
server2      1526/tcp
```

This entry makes **server2** known as the service name for TCP port 1526. A database server can then use this port to service connection requests. Figure 6-5 on page 6-28 includes a sample **services** file.

For information about the **hosts** and **services** files, refer to your operating-system documentation.

On UNIX, the **hosts** and **services** files are in the **/etc** directory. The files must be present on each computer that runs an Informix client/server product, or on the NIS server if your network uses *Network Information Service* (NIS).

Warning: *On systems that use NIS, the* ***/etc/hosts*** *and* ***/etc/services*** *files are maintained on the NIS server. The* ***/etc/hosts*** *and* ***/etc/services*** *files that reside on your local computer might not be used and might not be up to date. To view the contents of the NIS files, enter the following commands on the command line:*

```
ypcat hosts
ypcat services
```

Multiple TCP/IP Ports

To take advantage of multiple ethernet cards, take the following actions:

- Make an entry in the **services** file for each port the database server will use, as in the following example:

  ```
  soc1        21/tcp
  soc2        22/tcp
  ```

  All ports in use for a single IP address must be unique. Separate ethernet cards can utilize the same or different port numbers. You might want to use the same port number on each ethernet card because you are connecting to the same database server. (In this scenario, the service name is the same.)

- Put one entry per ethernet card in the **hosts** file with a separate IP address, as in the following example:

  ```
  192.147.104.19        svc8
  192.147.104.20        svc81
  ```

- In the ONCONFIG configuration file, enter DBSERVERNAME for one of the ethernet cards and DBSERVERALIASES for the other ethernet card. The following lines show sample entries in the ONCONFIG file:

  ```
  DBSERVERNAME chicago1
  DBSERVERALIASES chicago2
  ```

- In the **sqlhosts** file, make one entry for each ethernet card. That is, make an entry for the DBSERVERNAME and another entry for the DBSERVERALIAS.

  ```
  chicago1 onsoctcp svc8 soc1
  chicago2 onsoctcp svc81 soc2
  ```

Once this configuration is in place, the application communicates through the ethernet card assigned to the **dbservername** that the **INFORMIXSERVER** environment variable provides.

IPX/SPX Connectivity Files

To configure the database server to use the IPX/SPX protocol on a UNIX network, you must purchase IPX/SPX software and install it on the database server computer. Your choice of IPX/SPX software depends on the operating system that you are using. For some operating systems, the IPX/SPX software is bundled with software products based on NetWare for UNIX or Portable NetWare. In addition, for each of the UNIX vendors that distributes IPX/SPX software, you might find a different set of configuration files.

For advice on how to set configuration files for these software products, consult the manuals that accompany your IPX/SPX software.

Network-Security Files

Informix products follow standard security procedures that are governed by information contained in the network-security files. For a client application to connect to a database server on a remote computer, the user of the client application must have a valid user ID on the remote computer.

The hosts.equiv File

The **hosts.equiv** file lists the remote hosts and users that are trusted by the computer on which the database server resides. Trusted users, and users who log in from trusted hosts, can access the computer without supplying a password. The operating system uses the **hosts.equiv** file to determine whether a user should be allowed access to the computer without specifying a password. Informix requires a **hosts.equiv** file for its default authentication policy.

If a client application supplies an invalid account name and password, the database server rejects the connection even if the **hosts.equiv** file contains an entry for the client computer. You should use the **hosts.equiv** file only for client applications that do not supply a user account or password. On UNIX, the **hosts.equiv** file is in the **/etc** directory. If you do not have a **hosts.equiv** file, you must create one.

On some networks, the host name that a remote host uses to connect to a particular computer might not be the same as the host name that the computer uses to refer to itself. For example, the network host name might contain the full domain name, as the following example shows:

```
viking.informix.com
```

By contrast, the computer might refer to itself with the local host name, as the following example shows:

```
viking
```

If this situation occurs, make sure that you specify both host name formats in the **host.equiv** file.

To determine whether a client is trusted, execute the following statement on the client computer:

```
rlogin hostname
```

If you log in successfully without receiving a password prompt, the client is a trusted computer.

As an alternative, an individual user can list hosts from which he or she can connect as a trusted user in the **.rhosts** file. This file resides in the user's home directory on the computer on which the database server resides.

The netrc Information

The **netrc** information is optional information that specifies identity data. A user who does not have authorization to access the database server or is not on a computer that is trusted by the database server can use this file to supply a name and password that are trusted. A user who has a different user account and password on a remote computer can also provide this information.

The **netrc** information resides in the **.netrc** file in the user's home directory. Use any standard text editor to prepare the **.netrc** file.

If you do not explicitly provide the user password in an application for a remote server (that is, through the USER clause of the CONNECT statement or the user name and password prompts in DB-Access), the client application looks for the user name and password in the **netrc** information. If the user has explicitly specified the password in the application, or if the database server is not remote, the **netrc** information is not consulted.

The database server uses the **netrc** information regardless of whether it uses the default authentication policy or a communications support module.

For information about the specific content of this file, refer to your operating-system documentation.

User Impersonation

For certain client queries or operations, the database server must impersonate the client to run a process or program on behalf of the client. In order to impersonate the client, the database server must receive a password for each client connection. Clients can provide a user ID and password through the CONNECT statement or **netrc** information.

The following examples show how you can provide a password to impersonate a client.

File or Statement	Example
netrc information	`machine trngpc3 login bruce password im4golf`
CONNECT STATEMENT	`CONNECT TO ol_trngpc3 USER bruce USING "im4golf"`

CSM Configuration File

The **concsm.cfg** file describes the communication support module (CSM) and is required only if you use a CSM. An entry in the file is a single line and is limited to 1024 characters. After you describe the CSM in the **concsm.cfg** file, you can enable it in the options parameter of the **sqlhosts** file.

The **concsm.cfg** file resides in the **$INFORMIXDIR/etc** directory by default. If you want to store the file somewhere else, you can override the default location by setting the **INFORMIXCONCSMCFG** environment variable to the full pathname of the new location. For information on setting the environment variable **INFORMIXCONCSMCFG**, refer to the *Informix Guide to SQL: Reference*.

Format of the CSM Configuration File

The **concsm.cfg** file entry has the following format:

```
csmname(lib-paths, "csm-global-option",
    "csm-connection-options")
```

The *csmname* variable is the name that you assign to the communications support module. The *lib-paths* parameter has the following format:

```
"client=lib-path-clientsdk-csm, server=lib-path-server-csm"
```

The *lib-path-clientsdk-csm* is the full pathname, including the filename, of the shared library that is the CSM of the client, and the client applications use this CSM to communicate with the database server. The CSM is normally installed in **$INFORMIXDIR/lib/client/csm**.

The *lib-path-server-csm* is the full pathname, including the filename, of the shared library that is the CSM of the database server. The CSM is normally installed in **$INFORMIXDIR/lib/csm**, and the database server uses the CSM to communicate with the clients. The following restrictions apply to the CSM pathnames:

- Characters '=', '"' and ',' are not allowed to be part of the pathname.
- White spaces cannot be used between '=' and the pathname or between pathname and ',' or '"' unless the white spaces are part of the pathname.

The *lib-paths* parameter can alternatively have the following format:

```
        "lib-path-csm"
```

The *lib-path-csm* is the full pathname, including the filename, of the shared library that is the CSM. In this case, the same CSM is used by both the client applications and the database server.

The *csm-global* option is not used at this time for PWDCSM.

The *csm-connection-options* option can contain the following options.

Setting	Result
p=1	The password is mandatory for authentication.
p=0	The password is not mandatory. If the client provides it, the password is encrypted and used for authentication.

An unknown option placed in *csm-connection-options* results in a context initialization error.

You can put a null value in the *csm-connection-options* field. For Client SDK before Version 2.3, if the *csm-connection-options* field is null, the default behavior is p=1. For Client SDK, Version 2.3 and later, if the *csm-connection-options* field is null, the default behavior is p=0.

The concsm.cfg Entry for Password Encryption

The following two examples illustrate the two alternatives for parameters you must enter in the **concsm.cfg** file to define the Informix Password Communication Support Module:

```
PSWDCSM("client=/usr/informix/lib/client/csm/libixspw.so,
            server=/usr/informix/lib/csm/libixspw.so", "", "")

PSWDCSM("/usr/informix/lib/csm/libixspw.so", "", "")
```

The following example shows the *csm-connection-options* field set to 0, so no password is necessary:

```
PSWDCSM("/work/informix/csm/libixspw.so","","p=0")
```

The sqlhosts File

Informix client / server connectivity information, the ***sqlhosts*** *information*, contains information that enables a client application to find and connect to any Informix database server on the network.

For a detailed description of the **sqlhosts** information, refer to "The sqlhosts Information" on page 6-22.

On UNIX, the **sqlhosts** file resides, by default, in the **$INFORMIXDIR/etc** directory. As an alternative, you can set the **INFORMIXSQLHOSTS** environment variable to the full pathname and filename of a file that contains the **sqlhosts** file information. Each computer that hosts a database server or a client must have an **sqlhosts** file.

Each entry (each line) in the **sqlhosts** file contains the **sqlhosts** information for one database server or coserver. Use *white space* (spaces, tabs, or both) to separate the fields. You cannot include any spaces or tabs *within* a field. To put comments in the **sqlhosts** file, start a line with the comment character (#). You can also leave lines completely blank for readability. Additional syntax rules for each of the fields are provided in the following sections, which describe the entries in the **sqlhosts** file. Use any standard text editor to enter information in the **sqlhosts** file.

Figure 6-2 shows a sample **sqlhosts** file.

Figure 6-2
Sample sqlhosts File

dbservername	nettype	hostname	servicename	options
menlo	onipcshm	valley	menlo	
newyork	ontlitcp	hill	dynsrvr2	s=2,b=5120
sales	ontlispx	knight	sales	k=0,r=0
payroll	onsoctcp	dewar	py1	
asia	group	–	–	e=asia.3
asia.1	ontlitcp	node6	svc8	g=asia
asia.2	onsoctcp	node0	svc1	g=asia

The sqlhosts Information

The **sqlhosts** information in the **sqlhosts** file contains connectivity information for each database server. The **sqlhosts** information also contains definitions for groups. The database server looks up the connectivity information when you initialize the database server, when a client application connects to a database server, or when a database server connects to another database server.

The connectivity information for each database server includes four fields of required information and one optional field. The group information contains information in only three of its fields.

The five fields of connectivity information form one line in the UNIX **sqlhosts** file. The following table summarizes the fields used for the **sqlhosts** information.

UNIX Field Name	Description of Connectivity Information	Description of Group Information
dbservername	Database server name	Database server group name
nettype	Connection type	The word *group*
hostname	Host computer for the database server	*No information*. Use a hyphen as a placeholder in this field.
servicename	Alias for the port number	*No information*. Use a hyphen as a placeholder in this field.
options	Options that describe or limit the connection	Group options

If you install Informix Enterprise Gateway with DRDA in the same directory as the database server, your **sqlhosts** file also contains entries for the Gateway and non-Informix database servers. However, this manual covers only the entries for the database server. For information about other entries in the **sqlhosts** file, see the *Informix Enterprise Gateway with DRDA User Manual*.

Connectivity Information

The next section describes the connectivity information that is in each field of the **sqlhosts** file.

Database Server Name

The database server name field (**dbservername**) gives the name of the database server for which the connectivity information is being specified. Each database server across all of your associated networks must have a unique database server name. The **dbservername** field must match the name of a database server in the network, as specified by the DBSERVERNAME and DBSERVERALIASES configuration parameters in the ONCONFIG configuration file. For more information about these configuration parameters, refer to "ONCONFIG Parameters for Connectivity" on page 6-42.

The **dbservername** field can include any printable character other than an uppercase character, a field delimiter, a newline character, or a comment character. It is limited to 128 characters.

If the **sqlhosts** file has multiple entries with the same dbservername, only the first one is used.

The **sqlhosts** information must include the name of each coserver and the name of the group to which the coserver belongs.

The database server uses the **dbservername** field as the key to an index to look up the connectivity information in the remaining fields in the **sqlhosts** file. Specify a coserver name in this field. For a single-coserver system, the **sqlhosts** file should contain two entries:

- The database server name (for example, **asia**)
- The coserver name (for example, **asia.1**)

For a single-coserver system, set INFORMIXSERVER to either the database server name or the coserver name. See "Required Environment Variables" on page 3-12.

The field can also contain the name of a *dbserver group*. For more information about database server groups, refer to "Group Information" on page 6-37.

The Connection Type Field

The connection-type field (**nettype** on UNIX or PROTOCOL on Windows NT) describes the type of connection that should be made between the database server and the client application or another database server. The field is a series of eight letters composed of three subfields, as Figure 6-3 shows.

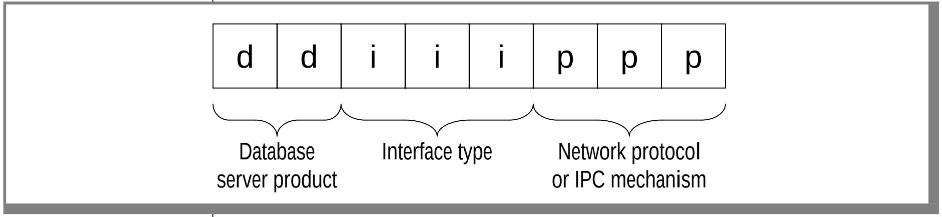


Figure 6-3
Format of the Connection-Type Field

The following sections describe the subfields of the connection-type field.

Database Server Product

The first two letters of the connection-type field represent the database server product.

Product Subfield	Product
on *or* ol	The database server
dr	Informix Enterprise Gateway with DRDA For information about DRDA, refer to the *Informix Enterprise Gateway with DRDA User Manual.*

Interface Type

The middle three letters of the connection-type field represent the network programming interface that enables communications. For more information, see "Network Programming Interface" on page 6-4.

Interface Subfield	Type of Interface
ipc	IPC (interprocess communications)
soc	Sockets
tli	TLI (transport layer interface)

Interprocess communications (IPC) are used only for communications between two processes running on the same computer.

Network Protocol

The final three letters of the connection-type field represent the network protocol or specific IPC mechanism.

Protocol Subfield	Type of Protocol
shm	Shared-memory communication
spx	IPX/SPX network protocol
str	Stream-pipe communication
tcp	TCP/IP network protocol

IPC connections use shared memory or stream pipes. The database server supports two network protocols: TCP/IP and IPX/SPX.

Figure 6-4 on page 6-26 summarizes the possible connection-type values for database server connections.

Figure 6-4
Summary of nettype Values

nettype value	**Description**	**Connection Type**
onipcshm	Shared-memory communication	IPC
onipcstr	Stream-pipe communication	IPC
ontlitcp	TLI with TCP/IP protocol	Network
onsoctcp	Sockets with TCP/IP protocol	Network
ontlispx	TLI with IPX/SPX protocol	Network

For information on the connection types for your platform, see "Connections That the Database Server Supports" on page 6-7.

Host Name Field

The host name field (**hostname**) contains the name of the computer where the database server resides. The name field can include any printable character other than a field delimiter, a newline character, or a comment character. The host name field is limited to 256 characters.

The following sections explain how client applications derive the values used in the host name field.

Network Communication with TCP/IP

When you use the TCP/IP network protocol, the host name field is a key to the **hosts** file, which provides the network address of the computer. The name that you use in the host name field must correspond to the name in the **hosts** file. In most cases, the host name in the **hosts** file is the same as the name of the computer. For information about the **hosts** file, refer to "TCP/IP Connectivity Files" on page 6-13.

In some situations, you might want to use the actual Internet IP address in the host name field. For information about using the IP address, refer to "IP Addresses for TCP/IP Connections" on page 6-38.

Shared-Memory and Stream-Pipe Communication

When you use shared memory or stream pipes for client/server communications, the **hostname** field must contain the actual host name of the computer on which the database server resides.

Network Communication with IPX/SPX

When you use the IPX/SPX network protocol, the **hostname** field must contain the name of the NetWare file server. The name of the NetWare file server is usually the UNIX *hostname* of the computer. However, this is not always the case. You might need to ask the NetWare administrator for the correct NetWare file-server names.

Tip: *NetWare installation and administration utilities might display the NetWare file-server name in capital letters; for example, VALLEY. In the* ***sqlhosts*** *file, you can enter the name in either uppercase or lowercase letters.*

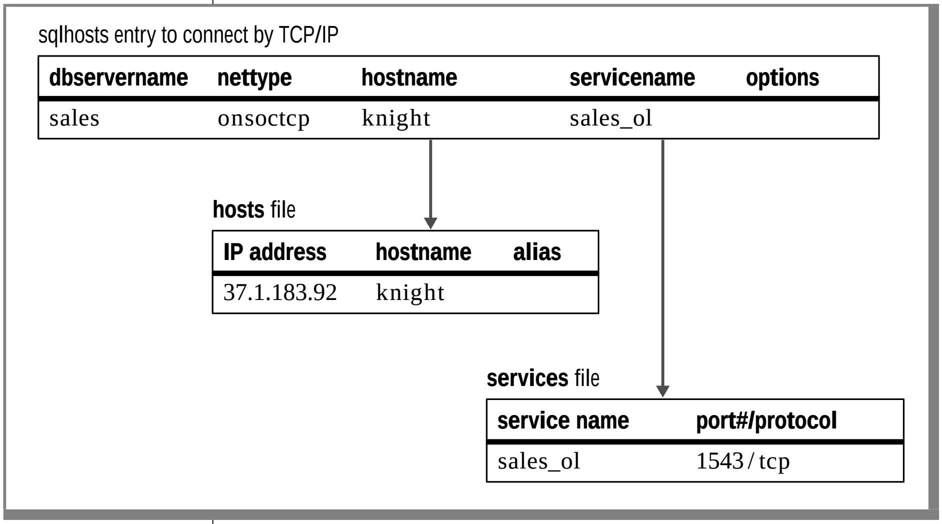
Service Name Field

The interpretation of the service name field (**servicename**) depends on the type of connection that the connection-type field (**nettype**) specifies.The service name field can include any printable character other than a field delimiter, a newline character, or a comment character. The service name field is limited to 128 characters.

Network Communication with TCP/IP

When you use the TCP/IP connection protocol, the service name field must correspond to a service name entry in the **services** file. The port number in the **services** file tells the network software how to find the database server on the specified host. It does not matter what service name you choose, as long as you agree on a name with the network administrator.

Figure 6-5 shows the relationship between the **sqlhosts** file and the **hosts** file, as well as the relationship of **sqlhosts** to the services file.



dbservername	nettype	hostname	servicename	options
sales	onsoctcp	knight	sales_ol	

IP address	hostname	alias
37.1.183.92	knight	

service name	port#/protocol
sales_ol	1543 / tcp

Figure 6-5
Relationship of sqlhosts File to hosts and services Files

In some cases, you might use the actual TCP listen-port number in the service name field. For information about using the port number, refer to "Port Numbers for TCP / IP Connections" on page 6-42.

Shared-Memory and Stream-Pipe Communication

When the **nettype** field specifies a shared-memory connection (`onipcshm`) or a stream-pipe connection (`onipcstr`), the database server uses the value in the **servicename** entry internally to create a file that supports the connection. For both `onipcshm` and `onipcstr` connections, the **servicename** can be any short group of letters that is unique in the environment of the host computer where the database server resides. Informix recommends that you use the **dbservername** as the **servicename** for stream-pipe connections.

Network Communication with IPX/SPX

A *service* on an IPX/SPX network is simply a program that is prepared to do work for you, such as the database server. For an IPX/SPX connection, the value in the **servicename** field can be an arbitrary string, but it must be unique among the names of services available on the IPX/SPX network. It is convenient to use the **dbservername** in the **servicename** field.

Options Field

The **options** field includes entries for the following features.

Option Name	Option Letter	Reference
Buffer size	b	page 6-30
Connection redirection	c	page 6-31
End of group	e	page 6-32
Group	g	page 6-33
Identifier	i	page 6-35
Keep-alive	k	page 6-35
Security	s (database server) r (client)	page 6-36
Communication support module	csm	page 6-32

When you change the values in the **options** field, those changes affect the next connection that a client application makes. You do not need to stop and restart the client application to allow the changes to take effect. However, a database server reads its own connectivity information *only* during initialization. If you change the options for the database server, you must reinitialize the database server to allow the changes to take effect.

Syntax Rules for the Options Field

Each item in the **options** field has the following format:

```
letter=value
```

You can combine several items in the **options** field, and you can include them in any order. The maximum length of the **options** field is 256 characters.

You can use either a comma or white space as the separator between options. You cannot use white space within an option.

The database server evaluates the **options** field as a series of columns. A comma or white space in the **options** field represents an end of a column. Client and database server applications check each column to determine whether the option is supported. If an option is not supported, you are not notified. It is merely ignored.

The following examples show both valid and invalid syntax.

Syntax	Valid	Comments
k=0,s=3,b=5120	Yes	Syntax is correct.
s=3,k=0 b=5120	Yes	Syntax is equivalent to the preceding entry. (White space is used of instead of a comma.)
k=s=0	No	You cannot combine entries.

Buffer-Size Option

Use the buffer-size option (`b=`*`value`*) to specify in bytes the size of the communications buffer space. The buffer-size option applies only to connections that use the TCP/IP network protocol. Other types of connections ignore the buffer-size setting. You can use this option when the default size is not efficient for a particular application. The default buffer size for the database server using TCP/IP is 4096 bytes.

Adjusting the buffer size allows you to use system and network resources more efficiently; however, if the buffer size is set too high, the user receives a connection-reject error because no memory can be allocated. For example, if you set `b=64000` on a system that has 1000 users, the system might require 64 megabytes of memory for the communications buffers. This setting might exhaust the memory resources of the computer.

On many operating systems, the maximum buffer size supported for TCP/IP is 16 kilobytes. To determine the maximum allowable buffer size, refer to the documentation for your operating system or contact the technical-support services for the vendor of your platform.

If your network includes several different types of computers, be particularly careful when you change the size of the communications buffer.

Tip: *Informix recommends that you use the default size for the communications buffer. If you choose to set the buffer size to a different value, set the client-side communications buffer and the database server-side communications buffer to the same size.*

Connection-Redirection Option

The connection-redirection option indicates the order in which the connection software chooses a coserver within a group.

Important: *The connection-redirection option is valid only in a dbserver group. For more information, see "Group Option" on page 6-33.*

By default, a client application is connected to the first coserver listed in the **sqlhosts** file. If the client cannot connect to the first database server, it attempts to connect to the second database server and so on.

If the connection-redirection option is on (`c=1`), the database server attempts to establish a connection in a random order among members of the dbserver group.

The following table shows the possible settings for the connection redirection option.

Setting	Result
c=0	Connect to a database server in the same order as listed in the group in the **sqlhosts** file.(This is the default setting).
c=1	Connect to a database server in a random order.

Communication Support Module Option

Use the communication support module (CSM) option to describe the CSM for each database server that uses a CSM. If you do not specify the CSM option, the database server uses the default authentication policy for that database server. You can specify the same CSM option setting for every database server described in the **sqlhosts** file, or you can specify a different CSM option or no CSM options for each **sqlhosts** entry.

The format of the CSM option is illustrated in the following example:

```
csm=(csmname,csm-connection-options)
```

The value of *csmname* must match a *csmname* entry in the **concsm.cfg** file. The *connection-options* parameter overrides the default *csm-connection* options specified in the **concsm.cfg** file. For information on the **concsm.cfg** file entry, refer to "CSM Configuration File" on page 6-19.

The following example specifies that the PSWDCSM communication support module will be used for the connection:

```
csm=(PSWDCSM)
```

For more information on the CSM, refer to "Database Server Connection" on page 6-5. For more information on the **concsm.cfg** file, refer to "CSM Configuration File" on page 6-19.

End of Group Option

Use this option to specify the ending database server name of a database server group. You can use this option only in a group entry. If you specify this option in an entry other than a database server group, it is ignored.

If no end-of-group option is specified for a group, the group members are assumed to be contiguous. The end of group is determined when an entry is reached that does not belong to the group or at the end of file, whichever comes first.

Group Option

When you define database server groups in the **sqlhosts** file key, you can use multiple related entries as one logical entity to establish or change client/server connections. Use the following steps to create database server groups.

To name a database server group

1. Specify the name of the database server group to which the **sqlhosts** entry belongs (up to 18 characters) in the DBSERVERNAME field.

   The database server group name can be the same as the initial DBSERVERNAME for the database server.
2. Place the keyword **group** in the connection type field.
3. The host name and service name fields are not used. Use dash (-) characters as null-field indicators for the unused fields. If you do not use options, you can omit the null-field indicators.

   The only options available for a database server group entry are as follows:

   ■ c = connection redirection

   ■ e = end of group

   ■ i = identifier option

To add coservers to a dbserver group

1. Specify the dbserver member that belongs to the dbserver group.
2. Indicate the nettype.
3. Identify the host name.
4. Identify the service name.
5. Identify the group with the group option

Important: *Database server groups cannot be nested inside other database server groups, and database server group members cannot belong to more than one group.*

Figure 6-6 on page 6-34 shows **sqlhosts** entries that define database server groups.

Figure 6-6
Database Server Groups in sqlhosts File

dbservername	nettype	hostname	servicename	options
asia	group	–	–	e=asia.3
asia.1	ontlitcp	node6	svc8	g=asia
asia.2	onsoctcp	node0	svc1	g=asia
usa.2	ontlispx	node9	sv2	
asia.3	onsoctcp	node1	svc9	g=asia
peru	group	–	–	
peru.1	ontlitcp	node4	svc4	
peru.2	ontlitcp	node5	svc5	g=peru
peru.3	ontlitcp	node7	svc6	
usa.1	onsoctcp	37.1.183.92	sales_ol	k=1, s=0
asia	group	–	–	e=asia.3
asia.1	ontlitcp	node6	svc8	g=asia
asia.2	onsoctcp	node0	svc1	g=asia
usa.2	ontlispx	node9	sv2	
asia.3	onsoctcp	node1	svc9	g=asia
peru	group	–	–	

The example in Figure 6-6 shows the following two groups: **asia** and **peru**. Group **asia** includes the following members:

- **asia.1**
- **asia.2**
- **asia.3**

Because group **asia** uses the end-of-group option (`e=asia.3`), the database server searches for group members until it reaches **asia.3**, so the group includes **usa.2**.

Because group **peru** does not use the end-of-group option, the database server continues to include all members until it reaches the end of file.

You can use the name of a database server group instead of the database server name in the following environment variables:

- **INFORMIXSERVER**

  The value of **INFORMIXSERVER** for a client application can be the name of a database server group. However, you cannot use a database server group name as the value of **INFORMIXSERVER** for a database server or database server utility.
- **DBPATH**

  DBPATH can contain the names of database server groups as dbservernames.

In addition, the group name can also be in the SQL CONNECT command.

Identifier Option

The identifier option assigns an identifying number to a database server group. The identifier must be a positive numeric integer and must be unique within your network environment.

Keep-Alive Option

The keep-alive option is a network option that TCP/IP uses. It does not affect other types of connections.

The letter *k* identifies keep-alive entries in the **options** field, as follows:

```
k=0     disable the keep-alive feature
k=1     enable the keep-alive feature
```

When a connected client and server are not exchanging data, the keep-alive option enables the network service to check the connection periodically. If the receiving end of the connection does not respond within the time specified by the parameters of your operating system, the connection is considered broken, and all resources related to the connection are released.

When the keep-alive option is enabled, the network service spends additional resources to check the connection.

When the keep-alive option is disabled, the network service does not check periodically whether the connection is still active. If the opposite end of the connection terminates unexpectedly without any notification, as when a PC reboots, for example, the network service might never detect that the connection is broken.

If you do not include the keep-alive option in the **options** field, the keep-alive feature is enabled by default. You can set this option on the database server side only, the client side only, or on both sides. For most cases, Informix recommends that you enable the keep-alive option.

Security Options

The security options let you control operating-system security-file lookups. The letter *s* identifies database server-side settings, and the letter *r* identifies client-side settings. You can set both options in the **options** field. A client ignores *s* settings, and the database server ignores *r* settings.

The following table shows the possible security option settings.

Setting	Result
r=0	Disables **netrc** lookup from the client side
r=1	Enables **netrc** lookup from the client side (default setting for the client side)
s=0	Disables both **hosts.equiv** and **.rhosts** lookup from the database server side
s=1	Enables only the **hosts.equiv** lookup from the database server side
s=2	Enables only the **.rhosts** lookup from the database server side
s=3	Enables both **hosts.equiv** and **.rhosts** lookup on the database server side (default setting for the database server side)

The security options let you control the way that a client (user) gains access to a database server. By default, an Informix database server uses the following information on the client computer to determine whether the client host computer is trusted:

- **hosts.equiv**
- **.rhosts** file

With the security options, you can specifically enable or disable the use of either or both files.

For example, if you want to prevent end users from specifying trusted hosts in the **.rhosts** file, you can set `s=1` in the **options** field of the **sqlhosts** filefor the database server to disable the **rhosts** lookup.

Group Information

The following section describes the fields of the **sqlhosts** file for groups.

Database Server Group

A database server group allows you to treat multiple related database servers entries as one logical entity to establish or change client/server connections. You can also use dbserver groups to simplify the redirection of connections to database servers. For more information on database server groups, see "Group Option" on page 6-33.

Group Keyword in the Connection-Type Field

The **group** keyword begins a list of coservers that are part of the database server. When a client connects to the database server using the database server name, the database server selects a coserver on which to establish the client connection.

Alternatives for TCP/IP Connections

The following sections describe some ways to bypass port and IP address lookups for TCP/IP connections.

IP Addresses for TCP/IP Connections

For TCP/IP connections (both TLI and sockets), you can use the actual Internet IP address in the **hostname** field instead of the host name or alias found in the **hosts** file. The IP address is always composed of four sets of one to three integers, separated by periods. Figure 6-7 shows sample IP addresses and host from a **hosts** file.

Figure 6-7
A Sample hosts File

Internet IP Address	Host Name	Host Alias(es)
555.12.12.12	smoke	
98.765.43.21	odyssey	
12.34.56.789	knight	sales

Using the IP address for **knight** from Figure 6-7, the following two **sqlhosts** entries are equivalent:

```
sales    ontlitcp    12.34.56.789    sales_ol
sales    ontlitcp    knight          sales_ol
```

Using an IP address might speed up connection time in some circumstances. However, because computers are usually known by their host name, using IP addresses in the host name field makes it less convenient to identify the computer with which an entry is associated.

You can find the IP address in the net address field of the **hosts** file, or you can use the UNIX **arp** or **ypmatch** command.

Wildcard Addressing for TCP/IP Connections

You can use wildcard addressing in the host name field when *both* of the following conditions are met:

- You are using TCP / IP connections.
- The computer where the database server resides has multiple network-interface cards (for example, three ethernet cards).

If the preceding conditions are met, you can use an asterisk (*) as a *wildcard* in the host name field that the database server uses. When you enter a wildcard in the host name field, the database server can accept connections at any valid IP address on its host computer.

Each IP address is associated with a unique host name. When a computer has multiple network-interface cards (NICs), as in Figure 6-8 on page 6-40, the **hosts** file must have an entry for each interface card. For example, the **hosts** file for the **texas** computer might include these entries.

NIC	Internet IP Address	Host Name
Card 1	123.45.67.81	texas1
Card 2	123.45.67.82	texas2

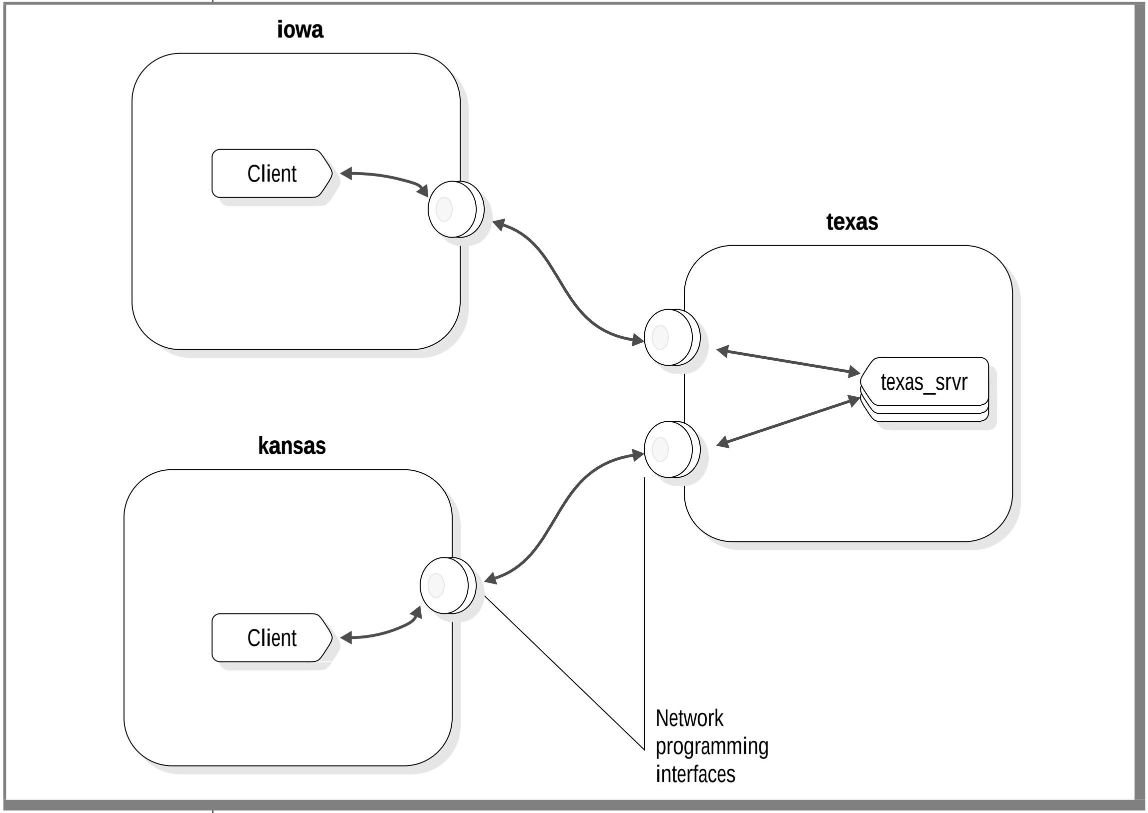
You can use the wildcard (*) alone or as a prefix for a host name or IP address, as shown in Figure 6-9 on page 6-41.

If the client application and database server share connectivity information (the **sqlhosts** file), you can specify both the wildcard and a host name or IP address in the **host name** field (for example, `*texas1` or `*123.45.67.81`). The client application ignores the wildcard and uses the host name (or IP address) to make the connection, and the database server uses the wildcard to accept a connection from any IP address.

The wildcard format allows the listen thread of the database server to wait for a client connection using the same service port number on each of the valid network-interface cards. However, waiting for connections at multiple IP addresses might require slightly more CPU time than waiting for connections with a specific host name or IP address.

Figure 6-8 shows a database server on a computer (**texas**) that has two network-interface cards. The two client sites use different network cards to communicate with the database server.

Figure 6-8
Using Multiple Network-Interface Cards



The connectivity information for the **texas_srvr** database server can be any of the entries in Figure 6-9.

Figure 6-9
Possible Connectivity Entries for the texas_srvr Database Server

database server name	connection type	host name	service name
texas_srvr	ontlitcp	*texas1	pd1_on
texas_srvr	ontlitcp	*123.45.67.81	pd1_on
texas_srvr	ontlitcp	*texas2	pd1_on
texas_srvr	ontlitcp	*123.45.67.82	pd1_on
texas_srvr	ontlitcp	*	pd1_on

Important: *You can include only one of these entries.*

If the connectivity information corresponds to any of the preceding lines, the **texas_srvr** database server can accept client connections from either of the network cards. The database server finds the wildcard in the **host name** field and ignores the explicit host name.

Tip: *For clarity and ease of maintenance, Informix recommends that you include a host name when you use the wildcard in the host name field (that is, use* `*host` *instead of simply* `*`*).*

The connectivity information used by a client application must contain an explicit host name or IP address. The client applications on **iowa** can use any of the following host names: `texas1`, `*texas1`, `123.45.67.81`, or `*123.45.67.81`. If there is an wildcard (*) in the **host name** field, the client application ignores it.

The client application on **kansas** can use any of the following host names: `texas2`, `*texas2`, `123.45.67.82`, or `*123.45.67.82`.

Port Numbers for TCP/IP Connections

For the TCP/IP network protocol, you can use the actual TCP listen port number in the **service name** field. The TCP port number is in the **port#** field of the **services** file.

Using the port number for the **sales** database server from the **services** file in Figure 6-5, an entry in **sqlhosts** could also be written as in the following example.

servername	nettype	hostname	servicename
sales	ontlitcp	knight	1536

Using the actual port number might save time when you make a connection in some circumstances. However, as with the IP address in the **host name** field, using the actual port number might make administration of the connectivity information less convenient.

ONCONFIG Parameters for Connectivity

When you initialize the database server, the initialization procedure uses parameter values from the ONCONFIG configuration file. The following ONCONFIG parameters are related to connectivity:

- DBSERVERNAME
- DBSERVERALIASES
- NETTYPE
- COSERVER

The next sections explain these configuration parameters.

COSERVER Configuration Parameter

The COSERVER parameter begins a coserver-specific section in the ONCONFIG file and specifies the numeric identification of a coserver. Each coserver requires a COSERVER section. When a client application connects to a coserver, it must specify a coserver name, which is formed by adding a dot and the coserver number to the dbservername, such as `river.1`, `river.2`, and so on. For more information, refer to the *Administrator's Reference*.

DBSERVERNAME Configuration Parameter

The DBSERVERNAME configuration parameter specifies a name, called the *dbservername*, for the database server. For example, to assign the value `nyc_research` to dbservername, use the following line in the ONCONFIG configuration file:

```
DBSERVERNAME nyc_research
```

When a client application connects to a database server, it must specify a dbservername. The **sqlhosts** information that is associated with the specified dbservername describes the type of connection that should be made.

The database server can use the coserver name, which is formed by concatenating the dbservername with the coserver number, as the key to the **sqlhosts** file. In this case, the **sqlhosts** information identifies the location of the connection coserver and the communication method that the client must use.

Client applications specify the name of the database server in one of the following places:

- In the **INFORMIXSERVER** environment variable
- In SQL statements such as CONNECT, DATABASE, CREATE TABLE, and ALTER TABLE, which let you specify a database environment
- In the **DBPATH** environment variable

DBSERVERALIASES Configuration Parameter

The DBSERVERALIASES parameter lets you assign additional dbservernames to the same database server. Figure 6-10 shows entries in an ONCONFIG configuration file that assign three dbservernames to the same database server instance.

Figure 6-10
Example of DBSERVERNAME and DBSERVERALIASES Parameters

```
DBSERVERNAME          sockets_srvr
DBSERVERALIASES       ipx_srvr,shm_srvr
```

The **sqlhosts** entries associated with the dbservernames from Figure 6-10 could include those shown in Figure 6-11. Because each dbservername has a corresponding entry in the **sqlhosts** file, you can associate multiple connection types with one database server.

Figure 6-11
Three Entries in the sqlhosts File for One Database Server

```
shm_srvr        onipcshm     my_host          my_shm
sockets_srvr    onsoctcp     my_host          port1
ipx_srvr        ontlispx     nw_file_server   ipx_srvr
```

Using the **sqlhosts** file shown in Figure 6-11, a client application uses the following statement to connect to the database server using shared-memory communication:

```
CONNECT TO '@shm_srvr'
```

A client application can initiate a TCP/IP sockets connection to the *same* database server using the following statement:

```
CONNECT TO '@sockets_srvr'
```

NETTYPE Configuration Parameter

The NETTYPE configuration parameter lets you adjust the number and type of virtual processors the database server uses for communication. Each type of network connection (ipcshm, ipcstr, ipcnmp, soctcp, tlitcp, and tlispx) can have a separate NETTYPE entry in the configuration file.

Although the NETTYPE parameter is not a required parameter, Informix recommends that you set NETTYPE if you use two or more connection types. After the database server has been running for some time, you can use the NETTYPE configuration parameter to tune the database server for better performance.

For more information about NETTYPE, refer to "Network Virtual Processors" on page 11-26. For additional information, regarding the impact of the NETTYPE configuration parameter, refer to the *Administrator's Reference*.

Environment Variables for Network Connections

The **INFORMIXCONTIME** (connect time) and **INFORMIXCONRETRY** (connect retry) environment variables are *client* environment variables that affect the behavior of the client when it is trying to connect to a database server. Use these environment variables to minimize connection errors caused by busy network traffic.

If the client application explicitly attaches to shared-memory segments, you might need to set **INFORMIXSHMBASE** (shared-memory base). For more information, refer to "How a Client Attaches to the Communications Portion" on page 13-12.

The **INFORMIXSERVER** environment variable allows you to specify a default dbservername to which your clients will connect.

For more information on environment variables, see the *Informix Guide to SQL: Reference*.

Examples of Client/Server Configurations

The next several sections show the correct entries in the **sqlhosts** filefor several client/server connections. The following examples are included:

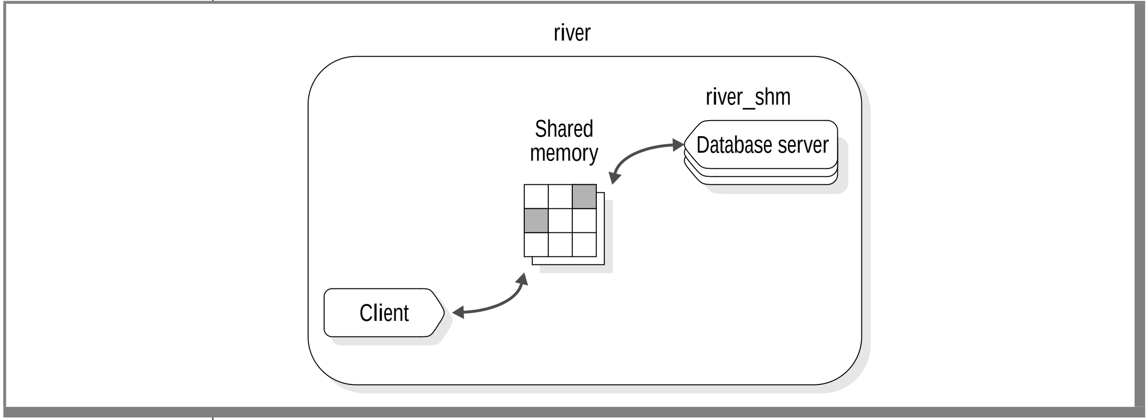
- Using a shared-memory connection
- Using a local-loopback connection
- Using a network connection
- Using multiple connection types
- Accessing multiple database servers

Important: *In the following examples, you can assume that the network-configuration files **hosts** and **services** have been correctly prepared even if they are not explicitly mentioned.*

Using a Shared-Memory Connection

Figure 6-12 shows a shared-memory connection on the computer named **river**.

Figure 6-12
A Shared-Memory Connection



The ONCONFIG configuration file for this installation includes the following line:

```
DBSERVERNAME river_shm
```

A correct entry for the **sqlhosts** file is as follows.

dbservername	nettype	hostname	servicename
river_shm	onipcshm	river	rivershm

The client application connects to this database server using the following statement:

```
CONNECT TO '@river_shm'
```

Because this is a shared-memory connection, no entries in network configuration files are required. For a shared-memory connection, you can choose arbitrary values for the **hostname** and **servicename** fields of the **sqlhosts** file .

For more information about shared-memory connections, refer to "How a Client Attaches to the Communications Portion" on page 13-12.

Using a Local-Loopback Connection

Figure 6-13 shows a local-loopback connection. The name of the host computer is **river**.

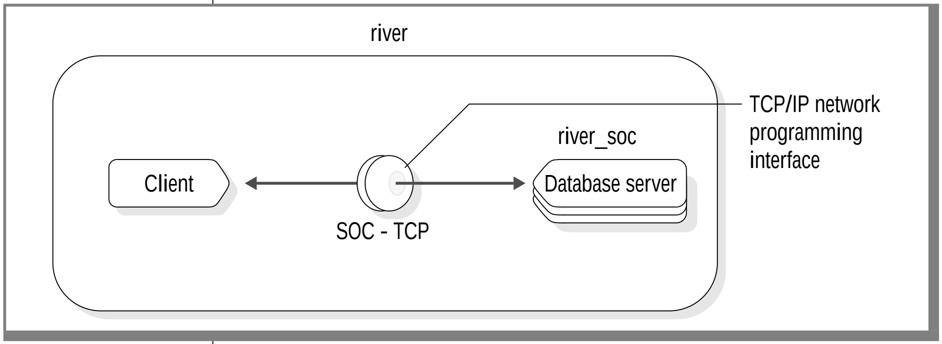


Figure 6-13
Local-Loopback Connection

The network connection in Figure 6-13 uses sockets and TCP / IP, so the correct entry for the **sqlhosts** file is as follows.

dbservername	nettype	hostname	servicename
river_soc	onsoctcp	river	riverol

If the network connection uses TLI instead of sockets, only the **nettype** entry in this example changes. In that case, the **nettype** entry is `ontlitcp` instead of `onsoctcp`.

The ONCONFIG file includes the following line:

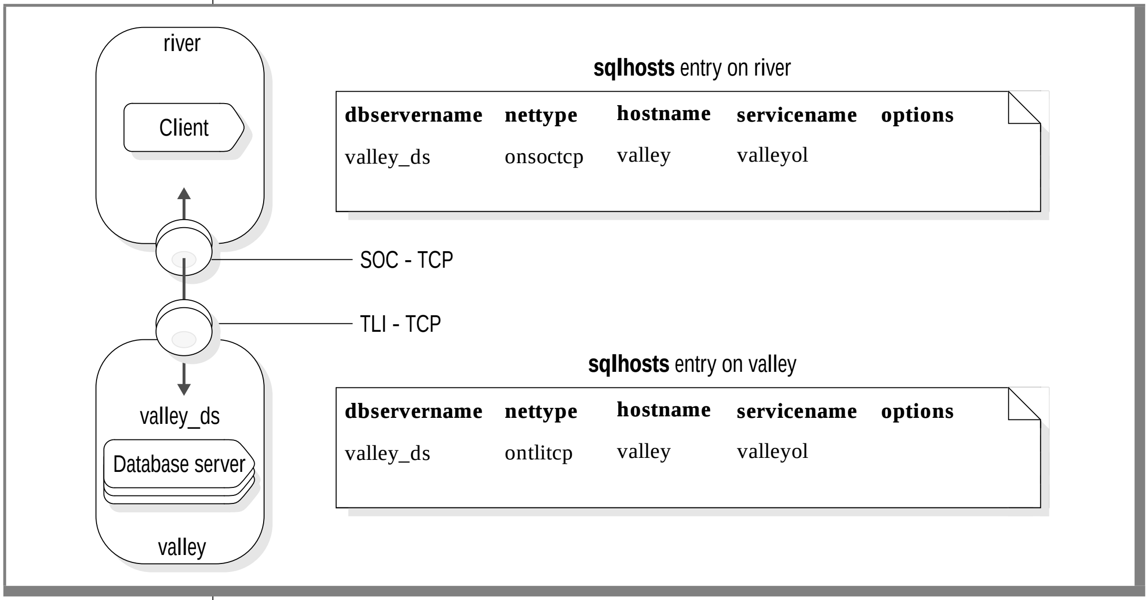
```
DBSERVERNAME river_soc
```

This example assumes that an entry for **river** is in the **hosts** file and an entry for **riverol** is in the **services** file.

Using a Network Connection

Figure 6-14 shows a configuration in which the client application resides on host **river** and the database server resides on host **valley**.

Figure 6-14
A Network Configuration



sqlhosts entry on river

dbservername	nettype	hostname	servicename	options
valley_ds	onsoctcp	valley	valleyol	

sqlhosts entry on valley

dbservername	nettype	hostname	servicename	options
valley_ds	ontlitcp	valley	valleyol	

An entry for the **valley_ds** database server is in the **sqlhosts** files on both computers. Each entry in the **sqlhosts** file on the computer where the database server resides has a corresponding entry on the computer where the client application resides.

Both computers are on the same TCP/IP network, but the host **river** uses sockets for its network programming interface, while the host **valley** uses TLI for its network programming interface. The **nettype** field must reflect the type of network programming interface used by the computer on which **sqlhosts** resides. In this example, the **nettype** field for the **valley_ds** database server on host **river** is `onsoctcp`, and the **nettype** field for the **valley_ds** database server on host **valley** is `ontlitcp`.

The sqlhosts File Entry for IPX/SPX

IPX/SPX software frequently provides TLI. If the configuration in Figure 6-14 on page 6-48 uses IPX/SPX instead of TCP/IP, the entry in the **sqlhosts** file on both computers is as follows.

dbservername	nettype	hostname	servicename
valley_us	ontlispx	valley_nw	valley_us

In this case, the **hostname** field contains the name of the NetWare file server. The **servicename** field contains a name that is unique on the IPX/SPX network and is the same as the dbservername.

Using Multiple Connection Types

A single instance of the database server can provide more than one type of connection. Figure 6-15 on page 6-50 illustrates such a configuration. The database server is on host **river**. Client A connects to the database server with a shared-memory connection because shared memory is fast. Client B must use a network connection because the client and server are on different computers.

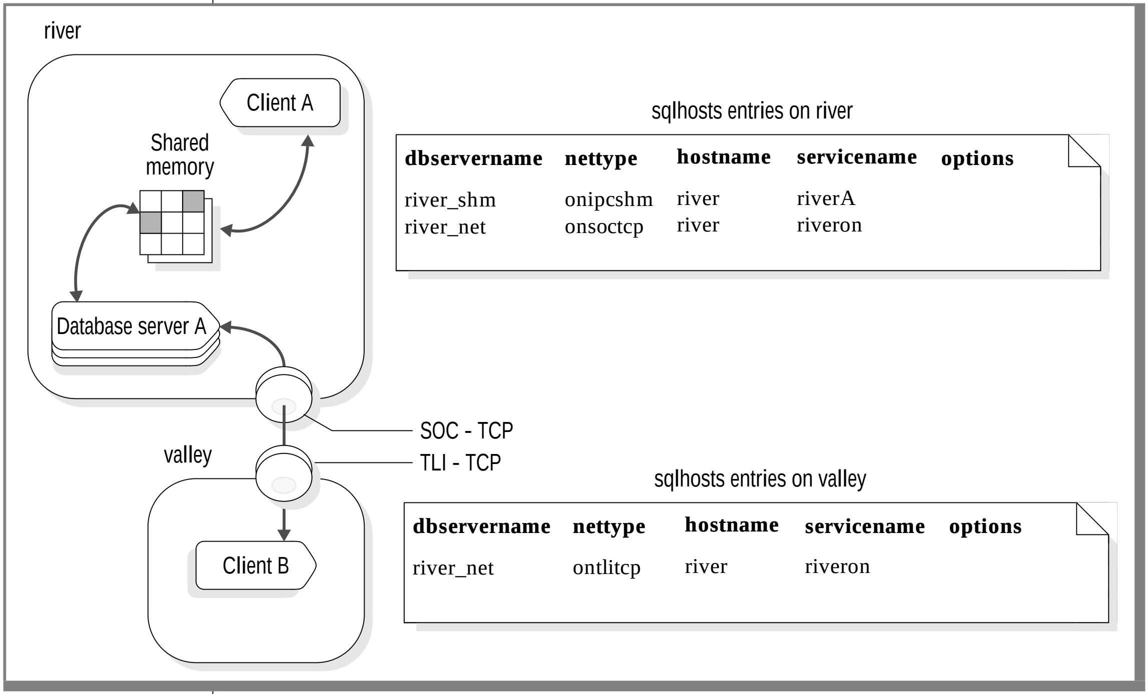
When you want the database server to accept more than one type of connection, you must take the following actions:

- Put DBSERVERNAME and DBSERVERALIASES entries in the ONCONFIG configuration file.
- Put an entry in the **sqlhosts** file for each database server/connection type pair.

For the configuration in Figure 6-15, the database server has two dbservernames: **river_net** and **river_shm**. The ONCONFIG configuration file includes the following entries:

```
DBSERVERNAME        river_net
DBSERVERALIASES     river_shm
```

Figure 6-15
A UNIX Configuration That Uses Multiple Connection Types



sqlhosts entries on river

dbservername	nettype	hostname	servicename	options
river_shm	onipcshm	river	riverA	
river_net	onsoctcp	river	riveron	

sqlhosts entries on valley

dbservername	nettype	hostname	servicename	options
river_net	ontlitcp	river	riveron	

The dbservername used by a client application determines the type of connection that is used. Client A uses the following statement to connect to the fully parallel-processing database server:

```
CONNECT TO '@river_shm'
```

In the **sqlhosts** file, the **nettype** associated with the name **river_shm** specifies a shared-memory connection, so this connection is a shared-memory connection.

Client B uses the following statement to connect to the database server:

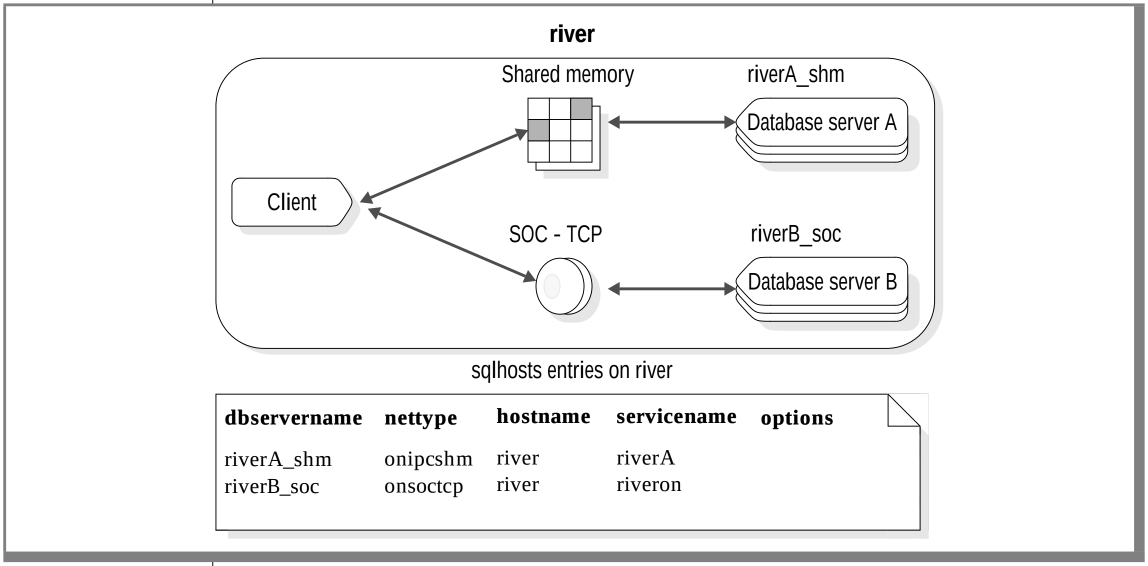
```
CONNECT TO '@river_net'
```

In the **sqlhosts** file, the **nettype** value associated with **river_net** specifies a network (TCP / IP) connection, so client B uses a network connection.

Accessing Multiple Database Servers

Figure 6-16 shows a configuration with two database servers on host **river**. When more than one database server is active on one computer, it is known as *multiple residency*. (For more information about multiple residency, see Chapter 7, “Multiple Residency.”)

Figure 6-16
Multiple Database Servers



dbservername	nettype	hostname	servicename	options
riverA_shm	onipcshm	river	riverA	
riverB_soc	onsoctcp	river	riveron	

For the configuration in Figure 6-16, you must prepare two ONCONFIG configuration files, one for database server **A** and the other for database server **B**. The **sqlhosts** file includes the connectivity information for both database servers.

The ONCONFIG configuration file for database server **A** includes the following line:

```
DBSERVERNAME    riverA_shm
```

The ONCONFIG configuration file for database server **B** includes the following line:

```
DBSERVERNAME    riverB_soc
```

Chapter

Multiple Residency

7

In This Chapter

You can use more than one database server in the following two ways:

- Run multiple instances of the database server on a single host computer
- Access several database servers over a network

When multiple database servers and their associated shared memory and disk structures coexist on a single computer, it is called *multiple residency.* This chapter covers the concept of multiple residency for the database server.

Benefits of Multiple Residency

Creating independent database server environments on the same computer allows you to perform the following actions:

- Separate production and development environments
- Isolate sensitive databases
- Test distributed data transactions on a single computer

When you use multiple residency, each database server has its own configuration file. Thus, you can create a configuration file for each database server that meets its special requirements for backups, shared-memory use, and tuning priorities.

You can separate production and development environments to protect the production system from the unpredictable nature of the development environment. You might also find it useful to isolate applications or databases that are critically important, either to increase security or to accommodate more frequent backups than most databases require.

If you are developing an application for use on a network, you can use local loopback to perform your distributed-data simulation and testing on a single computer. (See "Using a Local-Loopback Connection" on page 6-47.) Later, when a network is ready, you can use the application without changes to application source code.

However, running multiple database servers on the same computer is not as efficient as running one database server. You need to balance the advantages of separate database servers against the extra performance cost.

How Multiple Residency Works

Multiple residency is possible because the operating system can maintain separate areas in shared memory and on disk for each instance of the database server. Each instance of the database server passes a value to the operating system. This value, which is a function of the SERVERNUM configuration parameter, specifies the shared-memory address to which the database server process should attach. You must also specify a unique database server name and unique storage locations for each instance of the database server.

The Role of the ONCONFIG Environment Variable

The parameters in an ONCONFIG configuration file describe each instance of the database server. The **ONCONFIG** environment variable specifies the name of the current ONCONFIG configuration file. The following configuration parameters should have unique values for each database server:

- DBSERVERALIASES (if used)
- DBSERVERNAME
- MIRRORPATH
- MSGPATH
- ROOTPATH
- SERVERNUM

For instruction on setting these parameters, see Chapter 8, “Using Multiple Residency.”

The Role of the SERVERNUM Configuration Parameter

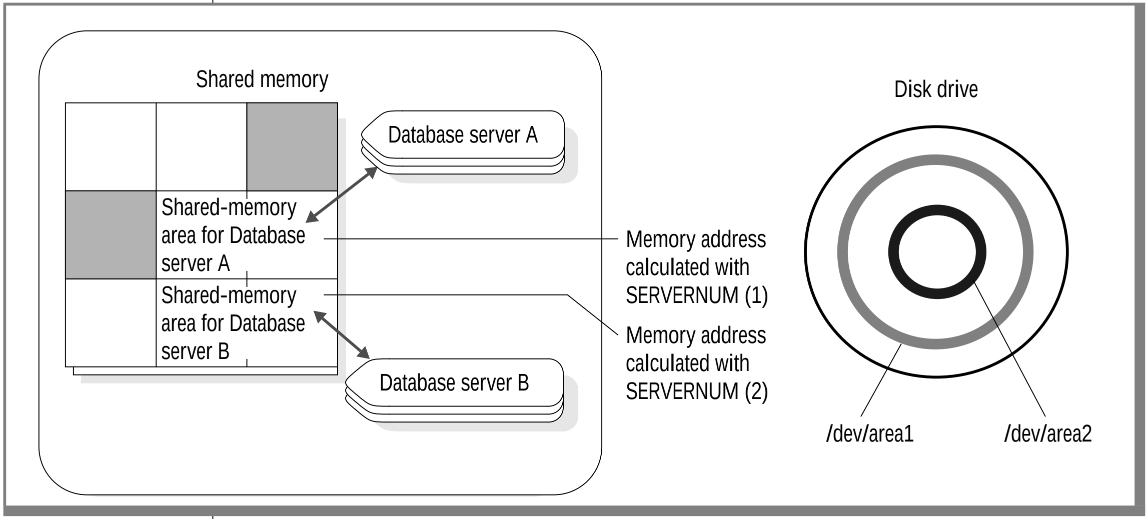
To maintain separation among the instances of database servers, you maintain multiple configuration files, each with a unique SERVERNUM value. When you initialize the database server, it reads the **ONCONFIG** environment variable for the name of its configuration file. Next, the database server reads its configuration file to obtain the value of its SERVERNUM configuration parameter. The database server then uses the SERVERNUM value to calculate the required shared-memory address.

For example, the ONCONFIG files for two database servers on a UNIX platform might include these parameters.

ONCONFIG file: onconfig.one	ONCONFIG file: onconfig.two
...	...
DBSERVERNAME dbsrvr_one	DBSERVERNAME dbsrvr_two
SERVERNUM 1	SERVERNUM 2
ROOTPATH /dev/area1	ROOTPATH /dev/area2
...	...

Figure 7-1 uses the configuration files shown in the preceding table to provide an example of multiple residency. Each database server has its own name, its own section of shared memory, and its own storage area on disk.

Figure 7-1
Separate Memory and Storage in Multiple Residency


Shared memory
Database server A
Shared-memory area for Database server A
Shared-memory area for Database server B
Database server B
Memory address calculated with SERVERNUM (1)
Memory address calculated with SERVERNUM (2)
Disk drive
/dev/area1
/dev/area2

Chapter

Using Multiple Residency

8

In This Chapter

This chapter describes how to use multiple database servers on the same computer. It includes the following topics:

- Planning for multiple residency
- Creating a database server instance

Before you set up multiple residency, you should install one database server as described in Chapter 3, "Installing and Configuring the Database Server."

Planning for Multiple Residency

When you plan for multiple residency on a computer, consider the following factors:

- Storage space

  Each database server must have its own unique storage space. You cannot use the same disk space for more than one instance of a database server. When you prepare an additional database server, you need to repeat some of the planning that you did to install the first database server. For example, you need to consider these questions:

  - Will you use buffered or unbuffered files? Will the unbuffered files share a disk partition with another application? (For more information on buffered and unbuffered files, see "Direct Disk Access" on page 1-12.)
  - Will you use mirroring? Where will the mirrors reside?
  - Where will the message log reside?
  - Can you dedicate a tape drive to this database server for its logical logs?
  - What kind of backups will you perform?

- Memory

  Each database server has its own memory. Can your computer handle the memory usage that an additional database server requires?

The following sections present the steps to create an additional database server, or multiple residency, on your computer.

Important: *Do not try to install another copy of the database server binary files. All instances of the same version of the database server on one computer share the same executable files.*

Creating a New Database Server

Perform the following steps manually to create an additional instance of the database server.

To create multiple residency of a database server

1. Prepare a new ONCONFIG configuration file.
2. Set the **ONCONFIG** environment variable to the new filename.
3. Edit the new ONCONFIG configuration file.
4. If needed, add a servicename to the **services** file or connection information to the NetWare server.
5. Update the **sqlhosts** file to include the dbservername(s) of the new database server.
6. Initialize disk space for the new database server.

To ensure proper backups, startup, and client connections

1. Prepare dbspace and backup schedules.
2. Modify the operating-system boot file.
3. Check the **INFORMIXSERVER** environment variables for users.

The following sections describe each of these steps.

Prepare a Configuration File

Each instance of the database server must have its own ONCONFIG configuration file. Make a copy of an ONCONFIG file that has the basic characteristics that you want for your new database server. Give the new file a name that you can easily associate with its function. For example, you might select the filename **onconfig.acct** to indicate the configuration file for a production system that contains accounting information.

Set the ONCONFIG Environment Variable

Set the **ONCONFIG** environment variable to the filename of the new ONCONFIG file. Specify only the filename, not the complete path.

Edit the New Configuration File

To edit the new ONCONFIG file, use a text editor.

In the new configuration file, you must change the following configuration parameters:

- SERVERNUM

  The SERVERNUM parameter specifies an integer (between 0 and 255) associated with a database server configuration. Each instance of a database server on the same host computer must have a unique SERVERNUM value. For more information, refer to "The Role of the SERVERNUM Configuration Parameter" on page 7-5.

- DBSERVERNAME

  The DBSERVERNAME parameter specifies the dbservername of a database server. Informix suggests that you choose a name that provides information about the database server, such as **ondev37** or ***hostname*****dev37**. For more information, refer to "DBSERVERNAME Configuration Parameter" on page 6-43.

- MSGPATH

  The MSGPATH parameter specifies the pathname of the message file for a database server. You should specify a unique pathname for the message file because database server messages do not include the dbservername. If multiple database servers use the same MSGPATH, you cannot identify the messages from separate database server instances. For example, if you name the database server **ondev37**, you might specify **/usr/informix/dev37.log** as the message log for this instance of the database server.

- ROOTPATH and ROOTOFFSET

  The ROOTPATH and ROOTOFFSET parameters together specify the location of the root dbspace for a database server. The root dbspace location must be unique for every database server configuration.

  If you put several root dbspaces in the same partition, you can use the same value for ROOTPATH. However, in that case, you must set ROOTOFFSET so that the combined values of ROOTSIZE and ROOTOFFSET define a unique portion of the partition. For more information about ROOTPATH and ROOTOFFSET, refer to the chapter on configuration parameters in the *Administrator's Reference*.

Tip: *You do not need to change ROOTNAME. Even if both database servers have the name* ***rootdbs*** *for their root dbspace, the dbspaces are unique because ROOTPATH specifies a unique location.*

You might also need to set the MIRRORPATH and MIRROROFFSET parameters. If the root dbspace is mirrored, the location of the root dbspace mirror must be unique. For information about setting MIRRORPATH, refer to "Steps Required for Mirroring Data" on page 26-3.

Add Connection Information

If you use the TCP/IP communication protocol, you might need to add an entry to the **services** file for the new database server instance. If you use the IPX/SPX communication protocol, you might need to modify the connection information for the NetWare server.

Update the sqlhosts File

The **sqlhosts** file must have an entry for each database server. If Informix products on other computers access this instance of the database server, the administrators on those computers must update their **sqlhosts** files. Chapter 6, "Client/Server Communications," discusses the preparation of the **sqlhosts** file.

If you plan to use TCP/IP network connections with an instance of a database server, the system network administrator must update the **hosts** and **services** files. If you use an IPX/SPX network, the NetWare administrator must update the NetWare file-server information. For information about these files, refer to "Network-Configuration Files" on page 6-13 and "IPX/SPX Connectivity Files" on page 6-16.

Initialize Disk Space

Before you initialize disk space, check the setting of your **ONCONFIG** environment variable. If you have not set it correctly, you might overwrite data from another database server. When you initialize disk space for a database server, the database server initializes the disk space specified in the current ONCONFIG configuration file.

Warning: *As you create new dbspaces for a database server, be sure to assign each chunk to a unique location on the device. The database server does not allow you to assign more than one chunk to the same location within a single database server environment, but it remains your responsibility as administrator to make sure chunks that belong to different database servers do not overwrite each other.*

Prepare Dbspace and Logical-Log Backup Environment

This section describes the effects of multiple residency on backups.

When you use multiple residency, you must maintain separate dbspace and logical-log backups for each database server instance. When you perform dbspace and logical-log backups with multiple residency, you need to be especially aware of device use.

If you can dedicate a tape drive to each database server, you can use the continuous logging option to back up your logical-log files. Otherwise, you must plan your dbspace and logical-log backup schedules carefully so that use of a device for one database server instance does not cause the other database server instance to wait. You must reset the ONCONFIG parameter each time that you switch your backup operations from one database server instance to the other.

Update the Operating-System Boot File

You can ask your system administrator to modify the system startup script ("Preparing the Startup and Shutdown Scripts" on page 3-18) so that each of your database server instances starts whenever the computer is rebooted (for example, after a power failure).

The startup script for a single database server instance should set the **INFORMIXDIR**, **PATH**, **ONCONFIG**, and **INFORMIXSERVER** environment variables and then execute **oninit**. To start a second instance of a database server, change the **ONCONFIG** and **INFORMIXSERVER** environment variables to point to the configuration file for the second database server and then execute **oninit** again. Do not change **INFORMIXDIR** or **PATH**.

Similarly, you can ask the system administrator to modify the shutdown script so that all instances of a database server shut down in a graceful manner.

Check INFORMIXSERVER Environment Variables for Users

If a new instance of a database should be the default database server, users need to reset the **INFORMIXSERVER** environment variable. Users might also need to update their **.informix** files.

If you use the **informix.rc** file to set environment variables for the users, you might need to update that file. The *Informix Guide to SQL: Reference* describes the **informix.rc** and **.informix** files.

Modes and Initialization

Section III

Chapter

9 Managing Database Server Operating Modes

In This Chapter

This chapter introduces you to the database server operating modes and provides instructions on how to change the operating modes of the database server.

Database Server Operating Modes

You can determine the current database server mode by executing the **onstat** utility from the command line. The **onstat** header displays the mode.

The database server has three principal modes of operation, as Figure 9-1 on page 9-4 illustrates.

Figure 9-1
Operating Modes

Operating Mode	Description
Off-line mode	When the database server is not running. No shared memory is allocated.
Quiescent, or administration, mode	When the database server processes are running and shared-memory resources are allocated, but the system does not allow database user access. Only the administrator (user **informix**) can access the database server.
On-line mode	Users can connect with the database server and perform all database activities. This is the normal operating mode of the database server. User **informix** or user **root** can use the command-line utilities to change many database server ONCONFIG parameter values while the database server is on-line.
Microkernel mode	Users can perform a cold restore with the database server in microkernel mode. For more information about cold restore, refer to the *Backup and Restore Guide*.

In addition, the database server can also be in one of the following modes:

- *Read-only mode* is used by the secondary database server in a high-availability data-replication pair. An application can query a database server that is in read-only mode, but the application cannot write to a read-only database.
- *Recovery mode* is transitory. It occurs when the database server performs fast recovery or recovers from a system archive or system restore. Recovery occurs during the change from Off-Line to Quiescent mode.
- *Shutdown mode* is transitory. It occurs when the database server is moving from on-line to quiescent mode or from on-line (or quiescent) to off-line mode. Current users access the system, but no new users are allowed access.

  Once shutdown mode is initiated, it cannot be cancelled.

Initializing Disk Space

If you are starting a new database server for the first time, or you want to remove all dbspaces and their associated data, use the following command to initialize the disk space and to bring the database server into on-line mode:

```
xctl -C oninit -iy
```

Warning: *When you execute this command, all existing data in the database server disk space is destroyed. Use the* ***-i*** *flag only when you are starting a new instance of the database server.*

Changing Database Server Operating Modes

This section describes how to change from one database server operating mode to another. Each section includes information on how to change operating modes with the **oninit** and **onmode** utilities.

In Extended Parallel Server, you can change from one database server operating mode to another while you are logged in on any node that supports a coserver. Use the **xctl** utility to execute the command on any coserver. For more information about **xctl**, refer to the utilities chapter in the *Administrator's Reference*.

Tip: *After you change the mode of your database server, execute the* ***xctl onstat*** *command to verify the current server status.*

Users Permitted to Change Modes

Only users who are logged in as **root** or **informix** can change the operating mode of the database server.

From Off-Line to Quiescent

When the database server changes from off-line mode to quiescent mode, the database server initializes shared memory.

When the database server is in quiescent mode, no sessions can gain access to the database server. In quiescent mode, any user can see status information.

To change a database server from off-line to quiescent, run the following command:

```
xctl -C oninit -s
```

From Off-Line to Microkernel

When you take the database server from off-line to microkernel mode, the database server is available to perform a cold restore of all stored data from a level-0 backup. To take the database server from off-line mode to microkernel mode for a cold restore, execute the following command:

```
xctl -C oninit -m
```

To verify that the database server is running, execute **xctl onstat** from the command line. The header on the **onstat** output displays the current operating mode.

Warning: *Perform a cold restore with ON-Bar only as a last resort to restore corrupted critical media. For information about performing cold restores from backup, refer to the "Backup and Restore Guide."*

From Off-Line to On-Line

When you take the database server from off-line mode to on-line mode, the database server initializes shared memory. When the database server is in on-line mode, it is available to all database server sessions.

Use one of the following methods to start the database server:

- To initialize all coservers defined in your ONCONFIG file, execute the following command:

  ```
  xctl -C oninit
  ```

- To initialize only a subset of coservers, execute the following command:

  ```
  xctl -X= -c nn -b oninit -X=
  ```

  The *nn* variable is the coserver number.

From Quiescent to On-Line

When you take the database server from quiescent mode to on-line mode, all sessions gain access.

If you have already taken the database server from on-line mode to quiescent mode and you are now returning the database server to on-line mode, any users who were interrupted in earlier processing must reselect their database and redeclare their cursors.

To take the database server from quiescent to on-line, execute the following command:

```
xctl onmode -m
```

Gracefully from On-Line to Quiescent

Take the database server gracefully from on-line mode to quiescent mode to restrict access to the database server without interrupting current processing.

After you perform this task, the database server sets a flag that prevents new sessions from gaining access to the database server. Current sessions are allowed to finish processing.

Once you initiate the mode change, it cannot be cancelled. During the mode change from on-line to quiescent, the database server is considered to be in Shutdown mode.

To bring the database server from on-line to quiescent gracefully, execute one of the following commands:

```
xctl onmode -s

xctl onmode -sy
```

Immediately from On-Line to Quiescent

Take the database server immediately from on-line mode to quiescent mode to restrict access to the database server as soon as possible. Work in progress can be lost.

The database server users receive either error message -459 indicating that the database server was shut down or error message -457 indicating that their session was unexpectedly terminated.

The database server performs proper cleanup on behalf of all sessions that the database server terminated. Active transactions are rolled back.

To move the database server to quiescent immediately, execute one of the following commands:

```
xctl onmode -u
```

```
xctl onmode -uy
```

From Any Mode Immediately to Off-Line

You can take the database server immediately from any mode to off-line mode. After you take the database server to off-line mode, reinitialize shared memory by taking the database server to quiescent or on-line mode. When you reinitialize shared memory, the database server performs a fast recovery to ensure that the data is logically consistent.

The database server users receive either error message -459 indicating that the database server was shut down or error message -457 indicating that their session was unexpectedly terminated.

After you take the database server to off-line mode, reinitialize shared memory by taking the database server to quiescent or on-line mode. When you reinitialize shared memory, the database server performs a fast recovery to ensure that the data is logically consistent.

The database server performs proper cleanup on behalf of all sessions that were terminated by the database server. Active transactions are rolled back.

To perform an immediate shutdown, perform one of the following commands:

- To shut down all coservers, execute:

  ```
  xctl onmode -k
  ```

  or

  ```
  xctl onmode -ky
  ```

- To shut down only one coserver, execute:

  ```
  xctl -X= -c nn onmode -ky -X=
  ```

The **-y** option eliminates the automatic prompt that confirms an immediate shutdown.

Chapter

Initializing the Database Server

10

In This Chapter

Initialization of the database server refers to two related activities: disk-space initialization and shared-memory initialization. This chapter defines the two types of initialization and describes the activities that take place during initialization.

Types of Initialization

Shared-memory initialization establishes the contents of database server shared memory as follows: internal tables, buffers, and the shared-memory communication area. Shared memory is initialized every time the database server starts up.

Disk-space initialization uses the values stored in the configuration file to create the initial chunk of the root dbspace on disk. When you initialize disk space, the database server automatically initializes shared memory as part of the process. Disk space is initialized the first time the database server starts up. It is only initialized thereafter during a cold recovery or at the request of the database server administrator.

Warning: *When you initialize disk space, you overwrite whatever is on that disk space. If you reinitialize disk space for an existing database server, all the data in the earlier database server becomes inaccessible and, in effect, is destroyed.*

Two key differences distinguish shared-memory initialization from disk-space initialization:

- Shared-memory initialization has no effect on disk-space allocation or layout. No data is destroyed.
- Shared-memory initialization performs fast recovery.

Initializing the Database Server

You must be logged in as **informix** or **root** to initialize the database server. The database server must be in off-line mode when you begin initialization.

You can use the **oninit** utility in conjunction with **xctl** to initialize the database server on multiple coservers. The options that you include in the **oninit** command determine the specific initialization procedure. For more information about these commands, refer to the utilities chapter in the *Administrator's Reference*.

Initialization Steps

Disk-space initialization always includes the initialization of shared memory. However, some activities that normally take place during shared-memory initialization, such as recording configuration changes, are not required during disk initialization because those activities are not relevant with a newly initialized disk.

Figure 10-1 shows the main tasks completed during the two types of initialization. The following sections discuss each step.

Figure 10-1
Initialization Steps

Shared-Memory Initialization	Disk Initialization
Process configuration file.	Process configuration file.
Create shared-memory segments.	Create shared-memory segments.
Initialize shared-memory structures.	Initialize shared-memory structures.
	Initialize disk space.
Start all required virtual processors.	Start all required virtual processors.
Make necessary conversions.	
Initiate fast recovery.	

(1 of 2)

Shared-Memory Initialization	Disk Initialization
Initiate a checkpoint.	Initiate a checkpoint.
Document configuration changes.	
Update **oncfg_*servername.servernum*** file.	Update **oncfg_*servername.servernum*** file.
Change to quiescent mode.	Change to quiescent mode.
Drop temporary tblspaces (optional).	
Set forced residency, if requested.	Set forced residency, if specified.
Change to on-line mode and return control to user.	Change to on-line mode and return control to user.
If the SMI tables are not current, update the tables.	Create SMI tables.

(2 of 2)

Process Configuration File

The database server uses configuration parameters to allocate shared-memory segments during initialization. If you change the size of shared memory by modifying a configuration-file parameter, you must take the database server to off-line mode and then reinitialize shared memory.

During initialization, the database server looks for configuration values in the following files, in this order:

1. If the **ONCONFIG** environment variable is set, the database server reads values from the **$INFORMIXDIR/etc/$ONCONFIG** file.

   If the **ONCONFIG** environment variable is set, but the database server cannot access the specified file, it returns an error message.
2. If the **ONCONFIG** environment variable is not set, the database server reads the configuration values from the **$INFORMIXDIR/etc/onconfig** file.
3. If you omit a configuration parameter in your ONCONFIG file, the database server reads the configuration values from the **$INFORMIXDIR/etc/onconfig.std** file.

Informix recommends that you *always* set the **ONCONFIG** environment variable before you initialize the database server. The default configuration files are intended as templates and not as functional configurations. For more information about the configuration file, refer to "Preparing the ONCONFIG Configuration File" on page 3-14.

The initialization process compares the values in the current configuration file with the previous values, if any, that are stored in the root dbspace reserved page, PAGE_CONFIG. For more information about PAGE_CONFIG, refer to the chapter on configuration parameters in the *Administrator's Reference*. When differences exist, the database server uses the values from the current ONCONFIG configuration file for initialization.

Create Shared-Memory Portions

The database server uses the configuration values to calculate the required size of the database server resident shared memory. In addition, the database server computes additional configuration requirements from internal values. Space requirements for overhead are calculated and stored.

To create shared memory, the database server acquires the shared-memory space from the operating system for three different types of memory:

- Resident portion, used for data buffers, buffer tables, and so on
- Virtual portion, used for most internal and user-session memory requirements
- IPC communication portion, used for IPC communication

  The database server allocates this portion of shared memory only if you configure an IPC shared-memory connection.

Next, the database server attaches shared-memory segments to its virtual address space and initializes shared-memory structures. For more information about shared-memory structures, refer to "Virtual Portion of Shared Memory" on page 13-25.

After initialization is complete and the database server is running, it can create additional shared-memory segments as needed. The database server creates segments in increments of the page size.

Initialize Shared-Memory Structures

After the database server attaches to shared memory, it clears the shared-memory space of uninitialized data. Next the database server lays out the shared-memory header information and initializes data in the shared-memory structures. For example, the database server lays out the space needed for the logical-log buffer, initializes the structures, and links together the three individual buffers that form the logical-log buffer. For more information about these structures, refer to the **onstat** utility in the *Administrator's Reference*.

After the database server remaps the shared-memory space, it registers the new starting addresses and sizes of each structure in the new shared-memory header.

During shared-memory initialization, disk structures and disk layout are not affected. The database server reads essential address information, such as the locations of the logical and physical logs, from disk and uses this information to update pointers in shared memory.

Initialize Disk Space

This procedure is performed only during disk-space initialization. After shared-memory structures are initialized, the database server begins initializing the disk. The database server initializes all the reserved pages that it maintains in the root dbspace on disk and writes PAGE_PZERO control information to the disk. For more information about PAGE_PZERO, refer to the information on reserved pages in the disk structures and storage chapter of the *Administrator's Reference*.

Start All Required Virtual Processors

The database server starts all the virtual processors that it needs. The parameters in the ONCONFIG file influence what processors are started. For example, the NETTYPE parameter can influence the number and type of processors started for making connections. For more information about virtual processors, refer to "Virtual Processors" on page 11-3.

Make Necessary Conversions

The database server checks its internal files. If the files are from an earlier version, it updates these files to the current format. For information about database conversion, refer to the *Informix Migration Guide*.

Initiate Fast Recovery

The database server checks if fast recovery is needed and, if so, initiates it. For more information about fast recovery, refer to "Fast Recovery" on page 24-13.

Fast recovery is not performed during disk-space initialization because there is not yet anything to recover.

Initiate a Checkpoint

After fast recovery executes, the database server initiates a checkpoint. As part of the checkpoint procedure, the database server writes a checkpoint-complete message in the message log. For more information about checkpoints, refer to "Checkpoints" on page 24-4.

The database server now moves to quiescent mode or on-line mode, depending on how you started the initialization process.

Document Configuration Changes

The database server compares the current values stored in the configuration file with the values previously stored in the root dbspace reserved page PAGE_CONFIG. When differences exist, the database server notes both values (old and new) in a message to the message log.

This task is not performed during disk-space initialization.

Create the oncfg_servername.servernum File

The database server creates the **oncfg_*servername.servernum*** file and updates it every time that you add or delete a dbspace, logical-log file, or chunk. You do not need to manipulate this file in any way, but you can see it listed in your **$INFORMIXDIR/etc** directory. The database server uses this file during a full-system restore. For more information about the **oncfg_*servername.servernum*** file, refer to the *Administrator's Reference*.

Drop Temporary Tblspaces

The database server searches through all dbspaces for temporary tblspaces. (If you use the **-p** option of **oninit** to initialize the database server, the database server skips this step.) These temporary tblspaces, if any, are tblspaces left by user processes that died prematurely and were unable to perform proper cleanup. The database server deletes any temporary tblspaces and reclaims the disk space. For more information about temporary tblspaces, refer to "Temporary Tables" on page 15-30.

This task is not performed during disk-space initialization.

Set Forced Residency If Specified

If the value of the RESIDENT configuration parameter is -1 or a number greater than 0, the database server tries to enforce residency of shared memory. If the host computer system does not support forced residency, the initialization procedure continues. Residency is not enforced, and the database server sends an error message to the message log. For more information about the RESIDENT configuration parameter, refer to the *Administrator's Reference*.

Return Control to User

After the previous steps are complete, the database server writes an `initialization complete` message in the message log. For more information about the message path and the MSGPATH configuration parameter, refer to the *Administrator's Reference*.

At this point, control returns to the user. Any error messages generated by the initialization procedure are displayed in one or more of the following locations:

- The command line
- The database server message log file, specified by the **MSGPATH** environment variable

Prepare SMI Tables

Even though the database server has returned control to the user, it has not finished its work. The database server now checks the *system-monitoring interface* (SMI) tables. If the SMI tables are not current, the database server updates the tables. If the SMI tables are not present, as is the case when the disk is initialized, the database server creates the tables. After the database server builds the SMI tables, it puts the message `sysmaster database built successfully` in the message-log file. For more information about SMI tables, refer to the chapter on the **sysmaster** database in the *Administrator's Reference*.

If you shut down the database server before it finishes building the SMI tables, the process of building the tables aborts. This condition does not damage the database server. The database server simply builds the SMI tables the next time that you bring the database server on-line. However, if you do not allow the SMI tables to finish building, you cannot run any queries against those tables, and you cannot use ON-Bar for dbspace or logical-log backups.

After the SMI tables have been created, the database server is ready for use. The database server runs until you stop it or, possibly, until a malfunction. Informix recommends that you *do not* try to stop the database server by killing a virtual processor or another database server process. For more information, refer to "Starting and Stopping Virtual Processors" on page 12-6.

Disk, Memory, and Process Management

Section IV

Chapter

Virtual Processors and Threads

11

In This Chapter

This chapter explains how the database server uses virtual processors and threads within virtual processors to improve performance. It explains the types of virtual processors and how threads run within the virtual processors.

Virtual Processors

Database server processes are called *virtual processors* because they function similarly to the way that a CPU functions in a computer. Just as a CPU runs multiple operating-system processes to service multiple users, a database server virtual processor runs multiple *threads* to service multiple SQL client applications.

A virtual processor is a process that the operating system schedules for processing.

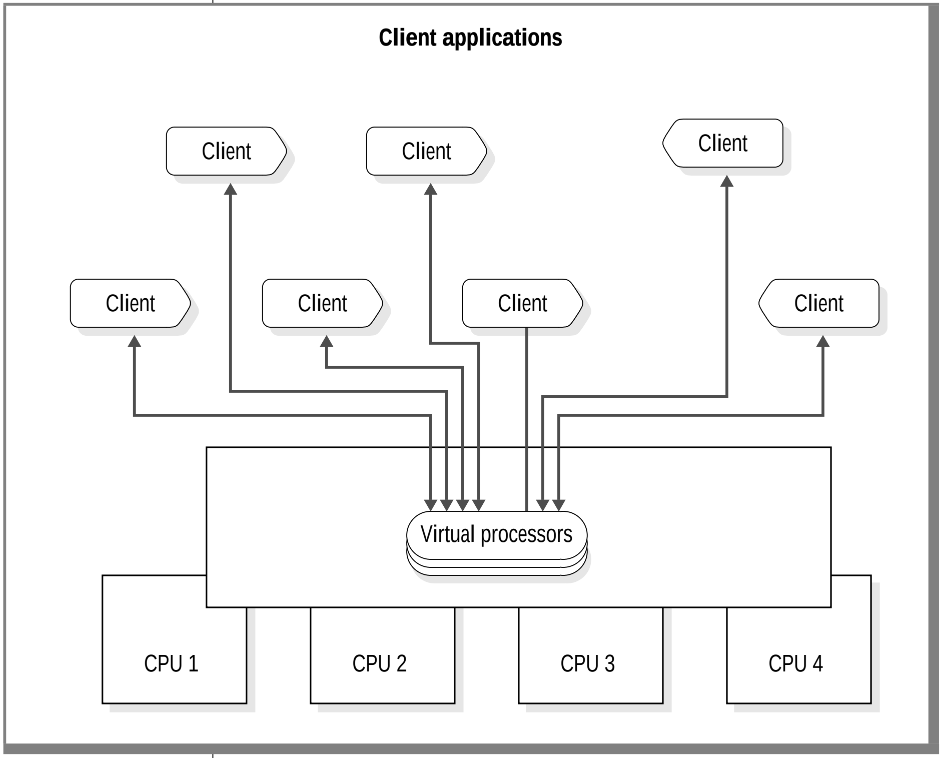


Figure 11-1
Virtual Processors

Figure 11-1 illustrates the relationship of client applications to virtual processors within a coserver.

Threads

A thread is a piece of work for a virtual processor in the same way that the virtual processor is a piece of work for the CPU. The virtual processor is a task that the operating system schedules for execution on the CPU; a database server thread is a task that the virtual processor schedules internally for processing. Threads are sometimes called *lightweight processes* because they are like processes, but they make fewer demands on the operating system.

Database server virtual processors are *multithreaded* because they run multiple concurrent threads.

A thread is a task that the virtual processor schedules internally for processing.

A virtual processor runs threads on behalf of SQL client applications (*session* threads) and also to satisfy internal requirements (*internal* threads). In most cases, for each connection by a client application, the database server runs one session thread. The database server runs internal threads to accomplish, among other things, database I/O, logging I/O, page cleaning, and administrative tasks. For cases in which the database server runs multiple session threads for a single client, refer to "Processing in Parallel" on page 11-8.

A *user thread* is a database server thread that services requests from client applications. User threads include session threads, called **sqlexec** threads, which are the primary threads that the database server runs to service client applications.

User threads also include a thread to service requests from the **onmode** utility, threads for recovery, and page-cleaner threads.

To display active user threads, use **onstat -u.** For more information on monitoring sessions and threads, refer to your *Performance Guide*.

Types of Virtual Processors

Figure 11-2 on page 11-6 shows the *classes* of virtual processors and the types of processing that they do. Each class of virtual processor is dedicated to processing certain types of threads.

Figure 11-2
Virtual-Processor Classes

Virtual-Processor Class	Category	Purpose
CPU	Central processing	Runs all session threads and some system threads. Runs thread for kernel asynchronous I/O where available. Can run a single poll thread, depending on configuration.
PIO	Disk I/O	Writes to the physical-log file (internal class) if it is in cooked disk space.
LIO	Disk I/O	Writes to the logical-log files (internal class) if they are in cooked disk space.
AIO	Disk I/O	Performs nonlogging disk I/O. If kernel asynchronous I/O is used, AIO virtual processors perform I/O to cooked disk spaces.
SHM	Network	Performs shared memory communication.
TLI	Network	Uses the Transport Layer Interface (TLI) to perform network communication.
SOC	Network	Uses sockets to perform network communication.
OPT	Optical	Performs I/O to optical disk.
ADM	Administrative	Performs administrative functions.
FIF	FIFO I/O	Performs reads and inserts for high performance loading and unloading through FIFO (first-in-first-out) files.
MSC	Miscellaneous	Services requests for system calls that require a very large stack.
CSM	Communications Support Module	Performs communications support service operations.

Advantages of Virtual Processors

Compared to a database server process that services a single client application, the dynamic, multithreaded nature of a database server virtual processor provides the following advantages:

- Virtual processors can share processing.
- Virtual processors save memory and resources.
- Virtual processors can perform parallel processing.
- You can start additional virtual processors and terminate active CPU virtual processors while the database server is running.
- You can bind virtual processors to CPUs.

The following sections describe these advantages.

Sharing Processing

Virtual processors in the same class have identical code and share access to both data and processing queues in memory. Any virtual processor in a class can run any thread that belongs to that class.

Generally, the database server tries to keep a thread running on the same virtual processor because moving it to a different virtual processor can require some data from the memory of the processor to be transferred on the bus. When a thread is waiting to run, however, the database server can migrate the thread to another virtual processor because the benefit of balancing the processing load outweighs the amount of overhead incurred in transferring the data.

Shared processing within a class of virtual processors occurs automatically and is transparent to the database user.

Saving Memory and Resources

The database server is able to service a large number of clients with a small number of server processes compared to architectures that have one client process to one server process. It does so by running a thread, rather than a process, for each client.

Multithreading permits more efficient use of the operating-system resources because threads share the resources allocated to the virtual processor. All threads that a virtual processor runs have the same access to the virtual-processor memory, communication ports, and files. The virtual processor coordinates access to resources by the threads. Individual processes, on the other hand, each have a distinct set of resources, and when multiple processes require access to the same resources, the operating system must coordinate the access.

Generally, a virtual processor can switch from one thread to another faster than the operating system can switch from one process to another. When the operating system switches between processes, it must stop one process from running on the processor, save its current processing state (or context), and start another process. Both processes must enter and exit the operating-system kernel, and the contents of portions of physical memory might need to be replaced. Threads, on the other hand, share the same virtual memory and file descriptors. When a virtual processor switches from one thread to another, the switch is simply from one path of execution to another. The virtual processor, which is a process, continues to run on the CPU without interruption. For a description of how a virtual processor switches from one thread to another, refer to "Context Switching" on page 11-12.

Processing in Parallel

In the following cases, virtual processors of the CPU class can run multiple session threads, working in parallel, for a single client:

- Index building
- Sorting
- Recovery
- Scanning
- Joining
- Aggregation
- Grouping

Figure 11-3 illustrates parallel processing. When a client initiates index building, sorting, or logical recovery, the database server spawns multiple threads to work on the task in parallel, using as much of the computer resources as possible. While one thread is waiting for I/O, another can be working.

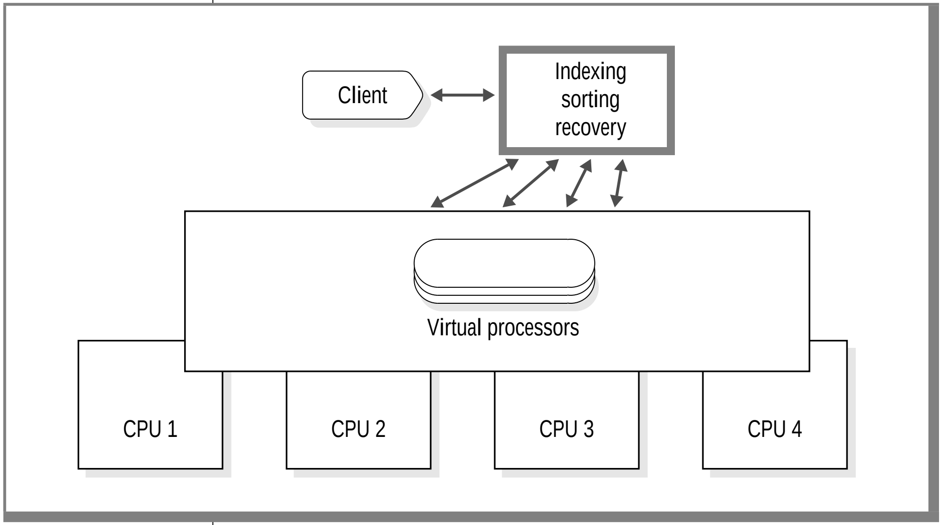


Figure 11-3
Parallel Processing

Adding and Dropping Virtual Processors in On-Line Mode

You can add virtual processors to meet increasing demands for service while the database server is running. For example, if the virtual processors of a class become compute bound or I/O bound (meaning that CPU work or I/O requests are accumulating faster than the current number of virtual processors can process them), you can start additional virtual processors for that class to distribute the processing load further.

You can add virtual processors for any of the classes while the database server is running. While the database server is running, you can drop virtual processors of the CPU class.

For information on how to add virtual processors while the database server is in on-line mode, refer to "Adding Virtual Processors in On-Line Mode" on page 12-6.

Binding Virtual Processors to CPUs

Some multiprocessor systems allow you to bind a process to a particular CPU. This feature is called *processor affinity*.

On multiprocessor computers for which the database server supports processor affinity, you can bind CPU virtual processors to specific CPUs in the computer. When you bind a CPU virtual processor to a CPU, the virtual processor runs exclusively on that CPU. This operation improves the performance of the virtual processor because it reduces the amount of switching between processes that the operating system must do. Binding CPU virtual processors to specific CPUs also enables you to isolate database work on specific processors on the computer, leaving the remaining processors free for other work. Only CPU virtual processors can be bound to CPUs.

For information on how to assign CPU virtual processors to hardware processors, refer to "Using Processor Affinity" on page 11-20.

How Virtual Processors Service Threads

At a given time, a virtual processor can run only one thread. A virtual processor services multiple threads concurrently by switching between them. A virtual processor runs a thread until it yields. When a thread yields, the virtual processor switches to the next thread that is ready to run. The virtual processor continues this process, eventually returning to the original thread when that thread is ready to continue. Some threads complete their work, and the virtual processor starts new threads to process new work. Because a virtual processor continually switches between threads, it can keep the CPU processing continually. The speed at which processing occurs produces the appearance that the virtual processor processes multiple tasks simultaneously and, in effect, it does.

Running multiple concurrent threads requires scheduling and synchronization to prevent one thread from interfering with the work of another. Virtual processors use the following structures and methods to coordinate concurrent processing by multiple threads:

- Control structures
- Context switching
- Stacks
- Queues
- Mutexes

This section describes how virtual processors use these structures and methods.

Control Structures

When a client connects to the database server, the database server creates a *session* structure, which is called a *session control block*, to hold information about the connection and the user. A session begins when a client connects to the database server, and it ends when the connection terminates.

Next, the database server creates a thread structure, which is called a *thread-control block* (TCB) for the session, and initiates a primary thread (**sqlexec**) to process the client request. When a thread *yields*—that is, when it pauses and allows another thread to run—the virtual processor saves information about the state of the thread in the thread-control block. This information includes the content of the process system registers, the program counter (address of the next instruction to execute), and the stack pointer. This information constitutes the *context* of the thread.

In most cases, the database server runs one primary thread per session. In cases where it performs parallel processing, however, it creates multiple session threads for a single client and, likewise, multiple corresponding thread-control blocks.

Context Switching

A virtual processor switches from running one thread to running another one by *context switching*. The database server does not preempt a running thread, as the operating system does to a process, when a fixed amount of time (time-slice) expires. Instead, a thread yields at one of the following points:

- A predetermined point in the code
- When the thread can no longer execute until some condition is met

When the amount of processing required to complete a task would cause other threads to wait for an undue length of time, a thread yields at a predetermined point. The code for such long-running tasks includes calls to the yield function at strategic points in the processing. When a thread performs one of these tasks, it yields when it encounters a yield function call. Other threads in the ready queue then get a chance to run. When the original thread next gets a turn, it resumes executing code at the point immediately after the call to the yield function. Predetermined calls to the yield function allow the database server to interrupt threads at points that are most advantageous for performance.

A thread also yields when it can no longer continue its task until some condition occurs. For example, a thread yields when it is waiting for disk I/O to complete, when it is waiting for data from the client, or when it is waiting for a lock or other resource.

When a thread yields, the virtual processor saves its context in the thread-control block. Then the virtual processor selects a new thread to run from a queue of ready threads, loads the context of the new thread from its thread-control block, and begins executing at the new address in the program counter. Figure 11-4 illustrates how a virtual processor accomplishes a context switch.

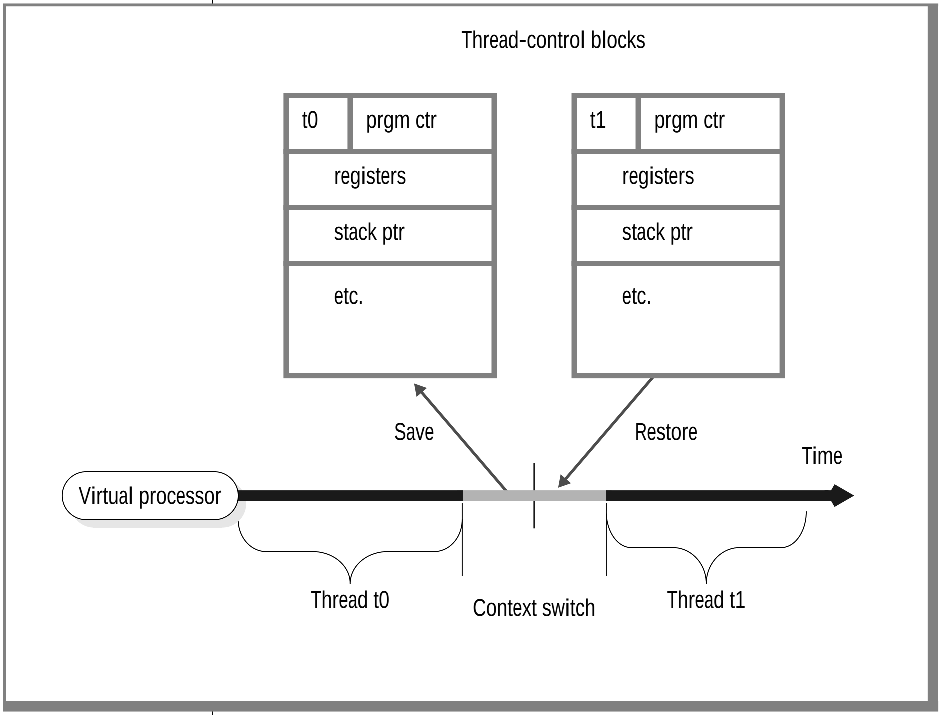


Figure 11-4
Context Switch: How a Virtual Processor Switches from One Thread to Another

Stacks

The database server allocates an area in the virtual portion of shared memory to store nonshared data for the functions that a thread executes. This area is called the *stack*. For information on how to set the size of the stack, refer to "Stacks" on page 13-31.

The stack enables a virtual processor to protect the nonshared data of a thread from being overwritten by other threads that concurrently execute the same code. For example, if several client applications concurrently perform SELECT statements, the session threads for each client execute many of the same functions in the code. If a thread did not have a private stack, one thread could overwrite local data that belongs to another thread within a function.

When a virtual processor switches to a new thread, it loads a stack pointer for that thread from a field in the thread-control block. The stack pointer stores the beginning address of the stack. The virtual processor can then specify offsets to the beginning address to access data within the stack. Figure 11-5 illustrates how a virtual processor uses the stack to segregate nonshared data for session threads.

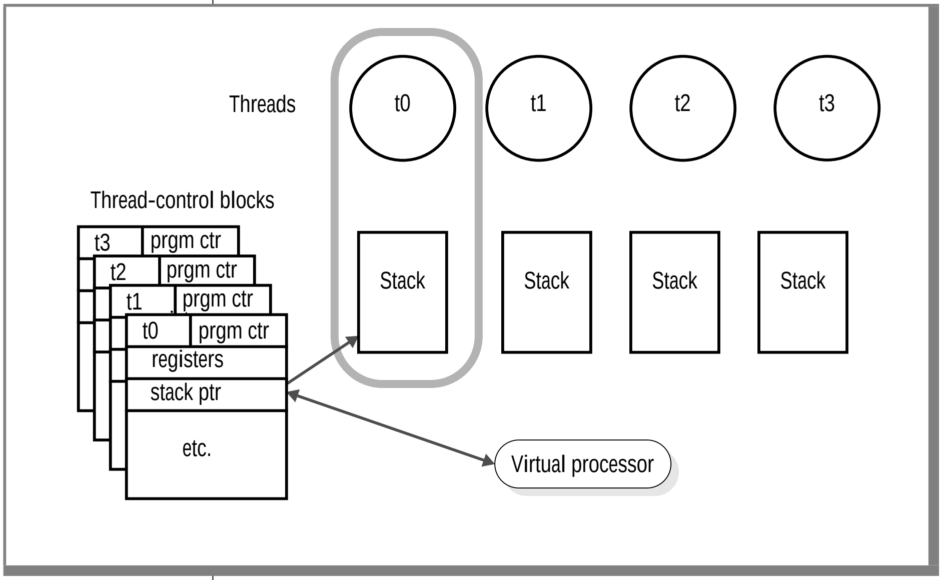


Figure 11-5
Virtual Processors Segregate Nonshared Data for Each User

Queues

The database server uses three types of queues to schedule the processing of multiple, concurrently running threads:

- Ready queues
- Sleep queues
- Wait queues

Virtual processors of the same class share queues. This fact, in part, enables a thread to migrate from one virtual processor in a class to another when necessary.

Ready Queues

Ready queues hold threads that are ready to run when the current (running) thread yields. When a thread yields, the virtual processor picks the next thread with the appropriate priority from the ready queue. Within the queue, the virtual processor processes threads that have the same priority on a first-in-first-out (FIFO) basis.

On a multiprocessor computer, if you notice that threads are accumulating in the ready queue for a class of virtual processors (indicating that work is accumulating faster than the virtual processor can process it), you can start additional virtual processors of that class to distribute the processing load. For information on how to monitor the ready queues, refer to "Monitoring Virtual Processors" on page 12-8. For information on how to add virtual processors while the database server is in on-line mode, refer to "Adding Virtual Processors in On-Line Mode" on page 12-6.

Sleep Queues

Sleep queues hold the contexts of threads that have no work to do at a particular time. A thread is put to sleep either for a specified period of time or *forever.*

The administration class (ADM) of virtual processors runs the system timer and special utility threads. Virtual processors in this class are created and run automatically. No configuration parameters impact this class of virtual processors.

The ADM virtual processor wakes up threads that have slept for the specified time. A thread that runs in the ADM virtual processor checks on sleeping threads at one-second intervals. If a sleeping thread has slept for its specified time, the ADM virtual processor moves it into the appropriate ready queue. A thread that is sleeping for a specified time can also be explicitly awakened by another thread.

A thread that is sleeping *forever* is awakened when it has more work to do. For example, when a thread that is running on a CPU virtual processor needs to access a disk, it issues an I/O request, places itself in a sleep queue for the CPU virtual processor, and yields. When the I/O thread notifies the CPU virtual processor that the I/O is complete, the CPU virtual processor schedules the original thread to continue processing by moving it from the sleep queue to a ready queue. Figure 11-6 illustrates how the database server threads are queued to perform database I/O.

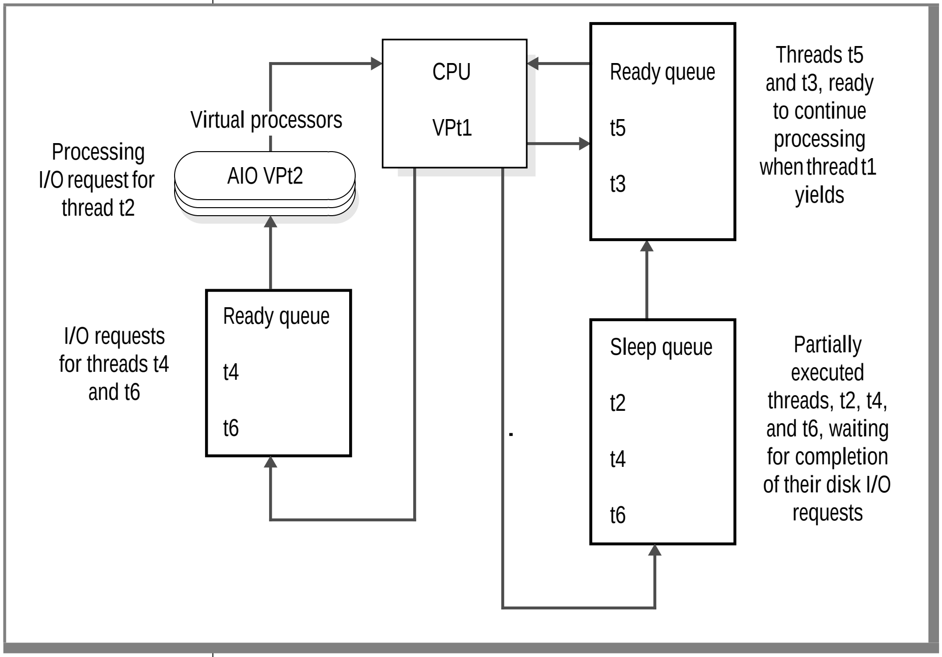

CPU
VPt1
Ready queue
t5
t3
Threads t5 and t3, ready to continue processing when thread t1 yields
Virtual processors
Processing I/O request for thread t2
AIO VPt2
I/O requests for threads t4 and t6
Ready queue
t4
t6
Sleep queue
t2
t4
t6
Partially executed threads, t2, t4, and t6, waiting for completion of their disk I/O requests

Figure 11-6
How Database Server Threads Are Queued to Perform Database I/O

Wait Queues

Wait queues hold threads that need to wait for a particular event before they can continue to run. For example, wait queues coordinate access to shared data by threads. When a user thread tries to acquire the logical-log latch but finds that the latch is held by another user, the thread that was denied access puts itself in the logical-log wait queue. When the thread that owns the lock is ready to release the latch, it checks for waiting threads and, if threads are waiting, it wakes up the next thread in the wait queue.

Mutexes

A mutex (*mut*ually *ex*clusive), also referred to as a *latch*, is a latching mechanism that the database server uses to synchronize access by multiple threads to shared resources. Mutexes are similar to semaphores, which some operating systems use to regulate access to shared data by multiple *processes*. However, mutexes permit a greater degree of parallelism than semaphores.

A mutex is a variable that is associated with a shared resource such as a buffer. A thread must acquire the mutex for a resource before it can access the resource. Other threads are excluded from accessing the resource until the owner releases it. A thread acquires a mutex, once a mutex becomes available, by setting it to an in-use state. The synchronization that mutexes provide ensures that only one thread at a time writes to an area of shared memory.

For information on monitoring mutexes, refer to "Monitoring Latches" on page 14-22.

Virtual-Processor Classes

A virtual processor of a given class can run only threads of that class. This section describes the types of threads, or the types of processing, that each class of virtual processor performs. It also explains how to determine the number of virtual processors that you need to run for each class.

CPU Virtual Processors

The CPU virtual processor runs all session threads (the threads that process requests from SQL client applications) and some internal threads. Internal threads perform services that are internal to the database server. For example, a thread that listens for connection requests from client applications is an internal thread.

Determining the Number of CPU Virtual Processors Needed

The right number of CPU virtual processors is the number at which they are all kept busy but not so busy that they cannot keep pace with incoming requests. You should not allocate more CPU virtual processors than the number of hardware processors in the computer.

The NUMCPUVPS configuration parameter allows you to specify the number of CPU virtual processors that the database server starts initially. For information about the NUMCPUVPS configuration parameter, refer to the *Administrator's Reference*.

To evaluate the performance of the CPU virtual processors while the database server is running, repeat the following command at regular intervals over a set period of time:

```
onstat -g glo
```

If the accumulated *usercpu* and *syscpu* times, taken together, approach 100 percent of the actual elapsed time for the period of the test, add another CPU virtual processor if you have a CPU available to run it.

For an additional consideration in deciding how many CPU virtual processors you need, refer to "Running Poll Threads on CPU or Network Virtual Processors" on page 11-27.

Running on a Multiprocessor Computer

If you are running multiple CPU virtual processors on a multiprocessor computer, set the MULTIPROCESSOR parameter in the ONCONFIG file to 1. When you set MULTIPROCESSOR to 1, the database server performs locking in a manner that is appropriate for a multiprocessor computer. For information on setting multiprocessor mode, refer to the chapter on configuration parameters in the *Administrator's Reference*.

Running on a Single-Processor Computer

If you are running only one CPU virtual processor, set the MULTIPROCESSOR configuration parameter to 0 and set the SINGLE_CPU_VP parameter to 1.

Setting MULTIPROCESSOR to 0 enables the database server to bypass the locking that is required for multiple processes on a multiprocessor computer. For information on the MULTIPROCESSOR configuration parameter, refer to the *Administrator's Reference*.

Setting SINGLE_CPU_VP to 1 allows the database server to bypass some of the mutex calls that it normally makes when it runs multiple CPU virtual processors. For information on setting the SINGLE_CPU_VP parameter, refer to the *Administrator's Reference*.

Important: *Setting NUMCPUVPS to 1 and SINGLE_CPU_VP to 0 does not reduce the number of mutex calls, even though the database server starts only one CPU virtual processor. You must set SINGLE_CPU_VP to 1 to reduce the amount of latching that is performed when you run a single CPU virtual processor.*

Setting the SINGLE_CPU_VP parameter to 1 imposes an important restriction on the database server: only one CPU virtual processor is allowed and you cannot add CPU virtual processors while the database server is in on-line mode.

For more information, see "Adding Virtual Processors in On-Line Mode" on page 12-6.

Adding and Dropping CPU Virtual Processors in On-Line Mode

You can add or drop CPU class virtual processors while the database server is on-line. For instructions on how to do this, see "Adding Virtual Processors in On-Line Mode" on page 12-6 and "Monitoring Virtual Processors" on page 12-8.

Preventing Priority Aging

Some operating systems lower the priority of long-running processes as they accumulate processing time. This feature of the operating system is called *priority aging*. Priority aging can cause the performance of database server processes to decline over time. In some cases, however, the operating system allows you to disable this feature and keep long-running processes running at a high priority.

To determine if priority aging is available on your computer, check the machine notes file described in "Documentation Notes, Release Notes, Machine Notes" on page 12 of the Introduction.

If your operating system allows you to disable priority aging, you can disable it by setting the NOAGE parameter. For more information on the NOAGE configuration parameter, refer to the *Administrator's Reference*.

Using Processor Affinity

On some multiprocessor platforms that support *processor affinity*, you can assign virtual processors to specific CPUs. When you assign a virtual processor to a specific CPU, the virtual processor runs exclusively on that CPU.

You can set the AFF_SPROC and AFF_NPROCS parameters in the ONCONFIG file to implement processor affinity on multiprocessor computers that support it.

Setting Processor Affinity with the AFF_SPROC and AFF_NPROCS Parameters

Set the AFF_NPROCS parameter to the number of CPUs to which you want to assign CPU virtual processors. Do not set AFF_NPROCS to a number that is less than the number of CPU virtual processors you have allocated. The number of CPUs should not be less than the number of CPU virtual processors that you allocate.

Set the AFF_SPROC parameter to the number of the first CPU to which a CPU virtual processor should be assigned. The database server assigns CPU virtual processors to CPUs in serial fashion, starting with this processor. The first CPU is number 0. For example, if the computer has four CPUs and you set NUMCPUVPS to 3, AFF_SPROC to 1, and AFF_NPROCS to 3, the three CPU virtual processors are assigned to the second, third, and fourth CPUs, respectively.

The value of AFF_NPROCS plus the value of AFF_SPROC must be less than or equal to the number of CPUs. In the previous example, if you set AFF_SPROC to 2, the database server would display an error message because 3 (AFF_NPROCS) plus 2 (AFF_SPROC) equals 5, and only four CPUs are available.

Figure 11-7 illustrates the concept of processor affinity.

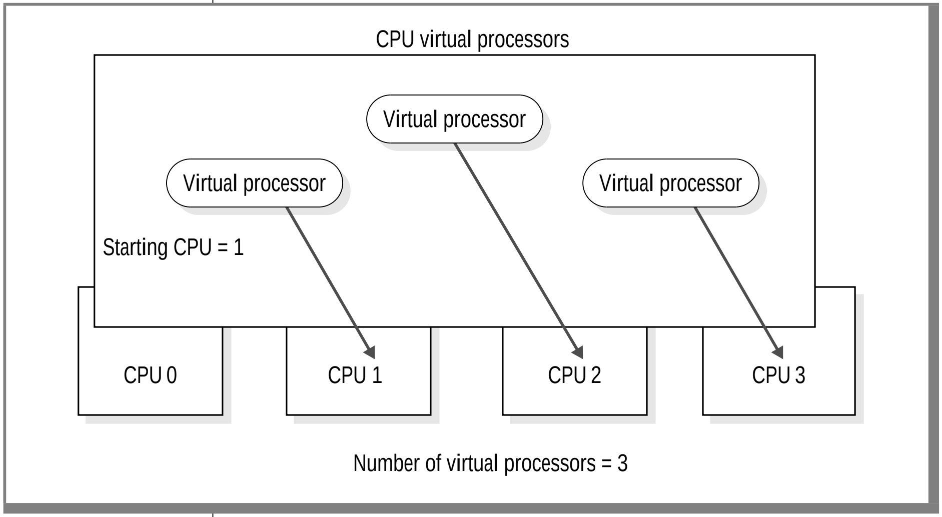


Figure 11-7
Processor Affinity

To see if processor affinity is supported on your platform, refer to the machine notes file.

Disk I/O Virtual Processors

The following classes of virtual processors perform disk I/O:

- PIO (physical-log I/O)
- LIO (logical-log I/O)
- AIO (asynchronous I/O)
- CPU (kernel-asynchronous I/O)

The PIO class performs all I/O to the physical-log file, and the LIO class performs all I/O to the logical-log files, *unless* those files reside in raw disk space and the database server has implemented kernel-asynchronous I/O.

On operating systems that do not support kernel-asynchronous I/O, the database server uses the AIO class of virtual processors to perform database I/O that is not related to physical or logical logging.

The database server uses the CPU class to perform kernel-asynchronous I/O (KAIO) when it is available on a platform. If the database server implements kernel-asynchronous I/O, a KAIO thread performs all I/O to raw disk space, including I/O to the physical and logical logs.

To find out if your platform supports kernel-asynchronous I/O, refer to the machine notes file.

For more information about nonlogging I/O, refer to "Asynchronous I/O" on page 11-24.

I/O Priorities

In general, the database server prioritizes disk I/O by assigning different types of I/O to different classes of virtual processors and by assigning priorities to the nonlogging I/O queues. Prioritizing ensures that a high-priority log I/O, for example, is never queued behind a write to a temporary file, which has a low priority. The database server prioritizes the different types of disk I/O that it performs, as Figure 11-8 shows.

Figure 11-8
How Database Server Prioritizes Disk I/O

Priority	Type of I/O	VP Class
1st	Logical-log I/O	CPU or LIO
2nd	Physical-log I/O	CPU or PIO
3rd	Database I/O	CPU or AIO
3rd	Page-cleaning I/O	CPU or AIO
3rd	Read-ahead I/O	CPU or AIO

Logical-Log I/O

The LIO class of virtual processors performs I/O to the logical-log files in the following cases:

- Kernel asynchronous I/O is not implemented.
- The logical-log files are in cooked disk space.

Only when kernel asynchronous I/O is implemented and the logical-log files are in raw disk space does the database server use a KAIO thread in the CPU virtual processor to perform I/O to the logical log.

The logical-log files store the data that enables the database server to roll back transactions and recover from system failures. I/O to the logical-log files is the highest priority disk I/O that the database server performs.

If the logical-log files are in a dbspace that *is not* mirrored, the database server runs only one LIO virtual processor. If the logical-log files are in a dbspace that *is* mirrored, the database server runs two LIO virtual processors. This class of virtual processors has no parameters associated with it.

Physical-Log I/O

The PIO class of virtual processors performs I/O to the physical-log file in the following cases:

- Kernel asynchronous I/O is not implemented.
- The physical-log file is stored in buffered-file chunks.

Only when kernel asynchronous I/O is implemented and the physical-log file is in raw disk space does the database server uses a KAIO thread in the CPU virtual processor to perform I/O to the physical log. The physical-log file stores *before-image*s of dbspace pages that have changed since the last *check-point*. (For more information on checkpoints, refer to "Checkpoints" on page 24-4.) At the start of recovery, prior to processing transactions from the logical log, the database server uses the physical-log file to restore before-images to dbspace pages that have changed since the last checkpoint. I/O to the physical-log file is the second-highest priority I/O after I/O to the logical-log files.

If the physical-log file is in a dbspace that *is not* mirrored, the database server runs only one PIO virtual processor. If the physical-log file is in a dbspace that *is* mirrored, the database server runs two PIO virtual processors. This class of virtual processors has no parameters associated with it.

Asynchronous I/O

The database server performs database I/O asynchronously, meaning that I/O is queued and performed independently of the thread that requests the I/O. Performing I/O asynchronously allows the thread that makes the request to continue working while the I/O is being performed.

The database server performs all database I/O asynchronously using one of the following facilities:

- AIO virtual processors
- Kernel-asynchronous I/O (KAIO) on platforms that support it.

Database I/O includes I/O for SQL statements, read-ahead, page cleaning, and checkpoints, as well as other I/O.

Kernel-Asynchronous I/O

The database server uses kernel-asynchronous I/O when the following conditions exist:

- The computer and operating system support it.
- A performance gain is realized.
- The I/O is to raw disk space.

The database server implements kernel-asynchronous I/O by running a KAIO thread on the CPU virtual processor. The KAIO thread performs I/O by making system calls to the operating system, which performs the I/O independently of the virtual processor. The KAIO thread can produce better performance for disk I/O than the AIO virtual processor can because it does not require a switch between the CPU and AIO virtual processors.

Informix implements kernel-asynchronous I/O when it ports the database server to a platform that supports this feature. The database server administrator does not configure kernel-asynchronous I/O. To see if kernel-asynchronous I/O is supported on your platform, see the machine notes file.

AIO Virtual Processors

If the platform does not support kernel-asynchronous I/O or if the I/O is to buffered-file chunks, the database server performs database I/O through the AIO class of virtual processors. All AIO virtual processors service all I/O requests equally within their class.

The database server assigns each disk chunk a queue, sometimes known as a *gfd queue*, based on the filename of the chunk. The database server orders I/O requests within a queue according to an algorithm that minimizes disk-head movement. The AIO virtual processors service queues that have work pending in round-robin fashion.

All other non-chunk I/O is queued in the *aio queue*.

Use the NUMAIOVPS parameter to specify the number of AIO virtual processors that the database server starts initially. For information about NUMAIOVPS, refer to chapter on configuration parameters the *Administrator's Reference*.

You can start additional AIO virtual processors while the database server is in on-line mode. For more information, refer to "Adding Virtual Processors in On-Line Mode" on page 12-6.

You cannot drop AIO virtual processors while the database server is in on-line mode.

Number of AIO Virtual Processors Needed

The goal in allocating AIO virtual processors 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 gfd queues are consistently short, it indicates that I/O to the disk devices is being processed as fast as the requests occur. The **onstat -g ioq** command allows you to monitor the length of the gfd queues for the AIO virtual processors. For more information, refer to "Monitoring Virtual Processors" on page 12-8.

If the database server implements kernel-asynchronous I/O on your platform, and all of your dbspaces are composed of raw disk space, one AIO virtual processor might be sufficient.

If the database server implements kernel-asynchronous I/O, but you are using some buffered files for chunks, allocate two AIO virtual processors per active dbspace that is composed of buffered file chunks. If kernel-asynchronous I/O is *not* implemented on your platform, allocate two AIO virtual processors for each disk that the database server accesses frequently.

Allocate enough AIO virtual processors to accommodate the peak number of I/O requests. Generally, it is not detrimental to allocate too many AIO virtual processors.

Network Virtual Processors

As explained in Chapter 6, "Client/Server Communications," a client can connect to the database server in the following ways:

- Through a network connection
- Through a pipe
- Through shared memory

The network connection can be made by a client on a remote computer or by a client on the local computer mimicking a connection from a remote computer (called a *local-loopback connection*).

Specifying Network Connections

In general, the DBSERVERNAME and DBSERVERALIASES parameters define dbservernames that have corresponding entries in the **sqlhosts** file. Each dbservername parameter in **sqlhosts** has a **nettype** entry that specifies an interface/protocol combination. The database server runs one or more *poll threads* for each unique **nettype** entry. For a description of the **nettype** field, refer to "The Connection Type Field" on page 6-24.

The NETTYPE configuration parameter provides optional configuration information for an interface/protocol combination. It allows you to allocate more than one poll thread for an interface/protocol combination and also designate the virtual-processor class (CPU or NET) on which the poll threads run. For a complete description of this configuration parameter, refer to the *Administrator's Reference*

Running Poll Threads on CPU or Network Virtual Processors

Poll threads can run either *in-line* on CPU virtual processors or, depending on the connection type, on network virtual processors. In general, and particularly on a single-processor computer, poll threads run more efficiently on CPU virtual processors. This might not be true, however, on a multiprocessor computer with a large number of remote clients.

The NETTYPE parameter has an optional entry, called `vp class`, that allows you to specify either CPU or NET, for CPU or network virtual-processor classes, respectively.

If you do not specify a virtual processor class for the interface / protocol combination (poll threads) associated with the DBSERVERNAME variable, the class defaults to CPU. The database server assumes that the interface / protocol combination associated with DBSERVERNAME is the primary interface / protocol combination and that it should be the most efficient.

For other interface / protocol combinations, if no `vp class` is specified, the default is NET.

While the database server is in on-line mode, you cannot drop a CPU virtual processor that is running a poll thread.

Number of Networking Virtual Processors Needed

Each poll thread requires a separate virtual processor, so you indirectly specify the number of networking virtual processors when you specify the number of poll threads for an interface / protocol combination and specify that they are to be run by the NET class. If you specify CPU for the `vp class`, you must allocate a sufficient number of CPU virtual processors to run the poll threads. If the database server does not have a CPU virtual processor to run a CPU poll thread, it starts a network virtual processor of the specified class to run it.

For most systems, one poll thread and consequently one virtual processor per network interface / protocol combination is sufficient. For systems with 200 or more network users, running additional network virtual processors might improve throughput. In this case, you need to experiment to determine the optimal number of virtual processors for each interface / protocol combination.

Listen and Poll Threads for the Client/Server Connection

When you start the database server, the **oninit** process starts an internal thread, called a *listen thread*, for each dbservername that you specify with the DBSERVERNAME and DBSERVERALIASES parameters in the ONCONFIG file. To specify a listen port for each of these dbservername entries, assign it a unique combination of **hostname** and **service name** entries in **sqlhosts**. For example, the **sqlhosts** file entry shown in Figure 11-9 causes the database server **soc_ol1** to start a listen thread for **port1** on the host, or network address, **myhost**.

Figure 11-9
A Listen Thread for Each Listen Port

dbservername	nettype	hostname	service name
soc_ol1	onsoctcp	myhost	port1

The listen thread opens the port and requests one of the poll threads for the specified interface / protocol combination to monitor the port for client requests. The poll thread runs either in the CPU virtual processor or in the network virtual processor for the connection that is being used. For information on the number of poll threads, refer to "Number of Networking Virtual Processors Needed" on page 11-27.

For information on how to specify whether the poll threads for an interface / protocol combination run in CPU or network virtual processors, refer to "Running Poll Threads on CPU or Network Virtual Processors" on page 11-27 and to the NETTYPE configuration parameter in the *Administrator's Reference*.

When a poll thread receives a connection request from a client, it passes the request to the listen thread for the port. The listen thread authenticates the user, establishes the connection to the database server, and starts an **sqlexec** thread, the session thread that performs the primary processing for the client. Figure 11-10 illustrates the roles of the listen and poll threads in establishing a connection with a client application.

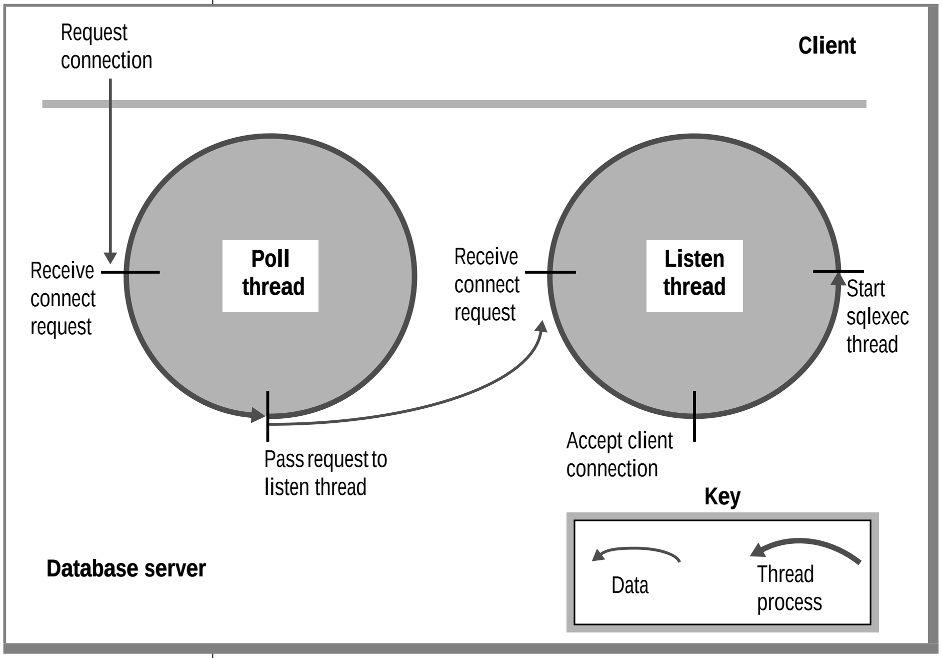


Figure 11-10
The Roles of the Poll and the Listen Threads in Connecting to a Client

A poll thread waits for requests from the client and places them in shared memory to be processed by the **sqlexec** thread. For network connections, the poll thread places the message in a queue in the shared-memory global pool. The poll thread then wakes up the **sqlexec** thread of the client to process the request. Whenever possible, the **sqlexec** thread writes directly back to the client without the help of the poll thread. In general, the poll thread reads data from the client, and the **sqlexec** thread sends data to the client.

For a shared-memory connection, the poll thread places the message in the communications portion of shared memory.

Figure 11-11 illustrates the basic tasks that the poll thread and the **sqlexec** thread perform in communicating with a client application.

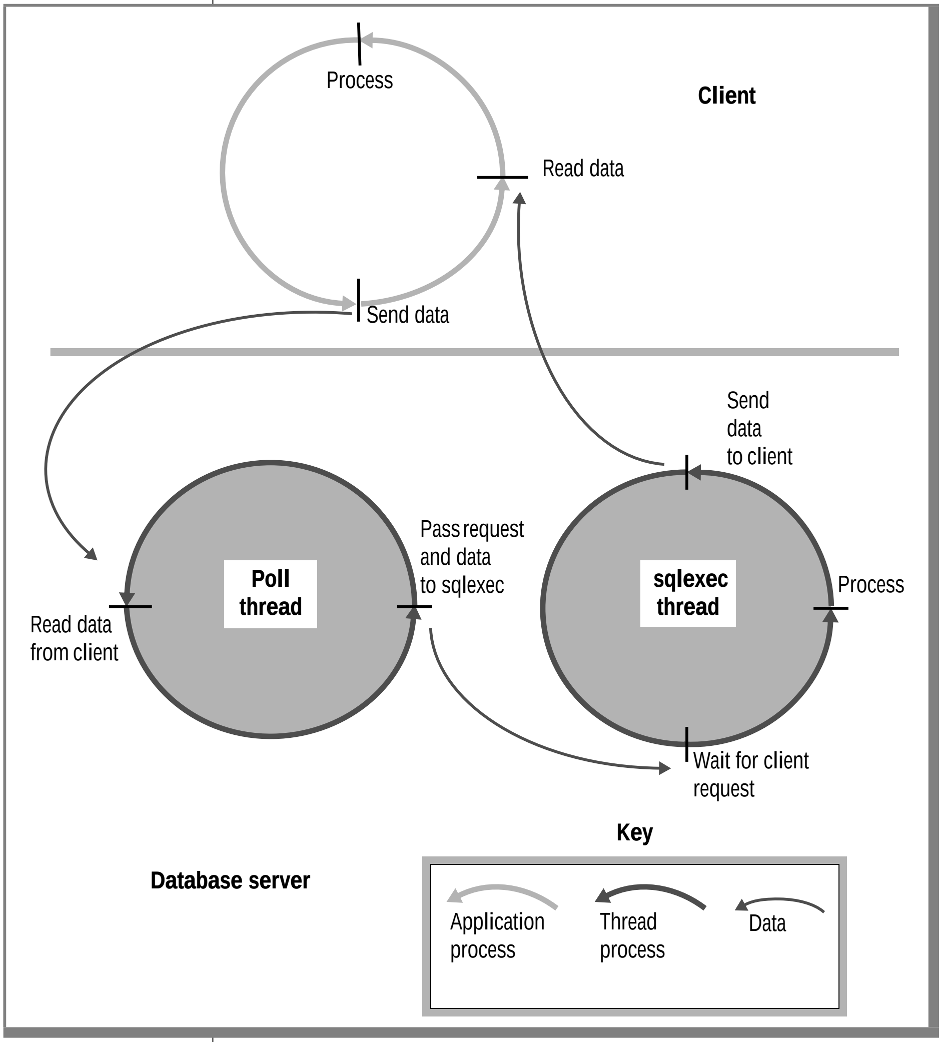


Figure 11-11
The Roles of the Poll and sqlexec Threads in Communicating with the Client Application

Starting Multiple Listen Threads

If the database server cannot service connection requests satisfactorily for a given interface / protocol combination with a single port and corresponding listen thread, you can improve service for connection requests in the following two ways:

- Add listen threads for additional ports.
- Add another network-interface card.

Adding Listen Threads

As stated previously, the database server starts a listen thread for each dbservername that you specify with the DBSERVERNAME and DBSERVERALIASES configuration parameters.

To add listen threads for additional ports, you must first use the DBSERVERALIASES parameter to specify dbservernames for each of the ports. For example, the DBSERVERALIASES parameter in Figure 11-12 defines two additional dbservernames, **soc_ol2** and **soc_ol3**, for the database server instance identified as **soc_ol1**.

```
DBSERVERNAME        soc_ol1
DBSERVERALIASES     soc_ol2,soc_ol3
```

Figure 11-12
Defining Multiple dbservernames for Multiple Connections of the Same Type

Once you define additional dbservernames for the database server, you must specify an interface / protocol combination and port for each of them in the **sqlhosts** file. Each port is identified by a unique combination of **hostname** and **servicename** entries. For example, the **sqlhosts** entries shown in Figure 11-13 on page 11-32 cause the database server to start three listen threads for the **onsoctcp** interface / protocol combination, one for each of the ports defined.

Figure 11-13
sqlhosts Entries to Listen to Multiple Ports for a Single Interface/Protocol Combination

dbservername	nettype	hostname	service name
soc_ol1	onsoctcp	myhost	port1
soc_ol2	onsoctcp	myhost	port2
soc_ol3	onsoctcp	myhost	port3

If you include a NETTYPE parameter for an interface / protocol combination, it applies to all the connections for that interface / protocol combination. In other words, if a NETTYPE parameter exists for **onsoctcp** in Figure 11-13, it applies to all of the connections shown. In this example, the database server runs one *poll* thread for the **onsoctcp** interface / protocol combination unless the NETTYPE parameter specifies more. For more information about entries in the **sqlhosts** file, refer to "Connectivity Files" on page 6-13.

Adding a Network-Interface Card

If the network-interface card for the host computer cannot service connection requests satisfactorily, or if you want to connect the database server to more than one network, you can add a network-interface card.

To support multiple network-interface cards, you must assign each card a unique **hostname** (network address) in **sqlhosts**. For example, using the same dbservernames shown in Figure 11-12, the **sqlhosts** file entries shown in Figure 11-14 cause the database server to start three listen threads for the same interface / protocol combination (as did the entries in Figure 11-13). In this case, however, two of the threads are listening to ports on one interface card (**myhost1**), and the third thread is listening to a port on the second interface card (**myhost2**).

Figure 11-14
Example of sqlhosts Entries to Support Two Network-Interface Cards for the onsoctcp Interface/Protocol Combination

dbservername	nettype	hostname	service name
soc_ol1	onsoctcp	myhost1	port1
soc_ol2	onsoctcp	myhost1	port2
soc_ol3	onsoctcp	myhost2	port1

First-In-First-Out Virtual Processor

The database server uses virtual processors of the FIF class to process high-performance loads and unloads through a FIFO (first-in-first-out) data file.

The operating system opens and checks for the end of file differently for FIFO files than for ordinary files. Unlike ordinary operating-system files, pipes do not have a 2-gigabyte size limitation.

For more information on using named pipes to load and unload tables, refer to the chapter on loading with external tables in the *Administrator's Reference*.

Communications Support Module Virtual Processor

The communications support module (CSM) class of virtual processors performs communications support service and communications support module functions.

The database server starts the same number of CSM virtual processors as the number of CPU virtual processors that it starts.

For more information on the communications support service, refer to Chapter 6, "Client / Server Communications."

Miscellaneous Virtual Processor

The miscellaneous virtual processor services requests for system calls that might require a very large stack, such as fetching information about the current user or the host-system name. Only one thread runs on this virtual processor; it executes with a stack of 128 kilobytes.

Chapter

Managing Virtual Processors

12

In This Chapter

This chapter describes how to set the configuration parameters that affect database server virtual processors. This chapter also tells you how to start and stop virtual processors.

For descriptions of the virtual-processor classes and for advice on how many virtual processors you should specify for each class, refer to Chapter 11, "Virtual Processors and Threads."

Setting Virtual-Processor Configuration Parameters

As **root** or user **informix**, you can set the configuration parameters for the database server virtual processors with a text editor.

To implement any changes that you make to configuration parameters, you must reinitialize shared memory. For information on how to reinitialize shared memory, refer to "Reinitializing Shared Memory" on page 14-11.

For more information on configuration parameters, refer to the *Administrator's Reference*.

Setting Virtual-Processor Configuration Parameters with a Text Editor

You can use a text editor program to set ONCONFIG parameters at any time. Use the editor to locate the parameter that you want to change, enter the new value, and rewrite the file to disk.

Figure 12-1 lists the ONCONFIG parameters that are used to configure virtual processors. For more information on how these parameters affect virtual processors, refer to "Virtual-Processor Classes" on page 11-17.

Figure 12-1
ONCONFIG Parameters for Configuring Virtual Processors

Parameter	Purpose
AFF_NPROCS	Specifies the number of CPUs to which CPU virtual processors will be assigned (multiprocessor computers only)
AFF_SPROC	Specifies the first CPU (of AFF_NPROCS) to which a CPU virtual processor will be assigned
MULTIPROCESSOR	Specifies that you are running on a multiprocessor computer
NETTYPE	Specifies parameters for network protocol threads (and virtual processors)
NOAGE	Specifies no priority aging of processes by the operating system
NUMAIOVPS	Specifies the number of AIO virtual processors
NUMCPUVPS	Specifies the number of CPU virtual processors
NUMFIFOVPS	Specifies the number of FIFO virtual processors
SINGLE_CPU_VP	Specifies that you are running a single CPU virtual processor

Specifying Virtual Processor Parameters for Uniprocessors or Symmetric Multiprocessors

If the nodes on your parallel-processing platform are uniprocessors, set the following parameters to the indicated values to configure each coserver for single-processor operation.

Parameter	Value
MULTIPROCESSOR	0
NUMCPUVPS	1
	1
	2

If your nodes are SMPs (symmetric multiprocessors), use the following parameters to control how the database server processes work on each multiprocessor. The exact values to use for these parameters depend on the number of CPUs in each SMP node:

- MULTIPROCESSOR
- NUMAIOVPS
- NUMCPUVPS
- SINGLE_CPU_VP

Disabling Priority Aging

Use the NOAGE parameter to disable process priority aging on platforms that allow this feature.

For recommended values for these database server parameters on your platform, refer to your machine notes file.

Starting and Stopping Virtual Processors

When you start the **oninit** process to start the database server, **oninit** starts the number and types of virtual processors that you have specified directly and indirectly. You configure virtual processors primarily through ONCONFIG parameters and, for network virtual processors, through parameters in the **sqlhosts** file. For descriptions of the virtual-processor classes, refer to "Virtual-Processor Classes" on page 11-17.

The database server allows you to start a maximum of 1000 virtual processors.

To terminate the database server and thereby terminate all virtual processors, use the **-k** option of the **onmode** utility. For more information on using **onmode -k**, refer to the utilities chapter of the *Administrator's Reference*.

Adding Virtual Processors in On-Line Mode

While the database server is in on-line mode, you can start additional virtual processors for the following classes: CPU, AIO, PIO, LIO, SHM, STR, TLI, and SOC.

To start these additional virtual processors, use the **-p** option of the **onmode** utility.

You can start additional virtual processors for FIF class to process high-performance loads and unloads through a FIFO (first-in-first-out) data file. FIFO files are often referred to as named pipes. For more information on using named pipes to load and unload tables, refer to the chapter on loading with external tables in the *Administrator's Reference*.

Adding Virtual Processors in On-Line Mode with onmode

Use the **-p** option of the **onmode** command to add virtual processors while the database server is in on-line mode. Specify the number of virtual processors that you want to add with a positive number that is greater than the number of virtual processors that are currently running. As an option, you can precede the number of virtual processors with a plus sign (+). Following the number, specify the virtual processor class in lowercase letters. For example, either of the following commands starts four additional virtual processors in the AIO class:

```
% onmode -p 4 aio
```

```
% onmode -p +4 aio
```

The **onmode** utility starts the additional virtual processors immediately.

You can add virtual processors to only one class at a time. To add virtual processors for another class, you must run **onmode** again.

Important: *You cannot add a CPU virtual processor with **onmode -p**.*

Adding Network Virtual Processors

When you add network virtual processors, you are adding poll threads, each of which requires its own virtual processor to run. If you attempt to add poll threads for a protocol while the database server is in on-line mode, and you have specified in the NETTYPE parameter that the poll threads run in the CPU class, the database server does not start the new poll threads if no CPU virtual processors are available to run them.

Monitoring Virtual Processors

Monitor the virtual processors to determine if the number of virtual processors configured for the database server is optimal for the current level of activity.

Monitoring Virtual Processors with Command-Line Utilities

You can use the following **onstat -g** options to monitor virtual processors:

- **glo**
- **ioq**
- **rea**

Use the **onstat -g glo** command to display information about each virtual processor that is currently running, as well as cumulative statistics for each virtual processor class. Figure 12-2 shows an example of the output from this option.

Figure 12-2
onstat -g glo Output

```
MT global info:
sessions threads  vps      lngspins
1        15       8        0

Virtual processor summary:
class    vps      usercpu          syscpu           total
 cpu       3      479.77           190.42           670.18
 aio       1        0.83             0.23             1.07
 pio       1        0.42             0.10             0.52
 lio       1        0.27             0.22             0.48
 soc       0        0.00             0.00             0.00
 tli       0        0.00             0.00             0.00
 shm       0        0.00             0.00             0.00
 adm       1        0.10             0.45             0.55
 opt       0        0.00             0.00             0.00
 msc       1        0.28             0.52             0.80
 adt       0        0.00             0.00             0.00
 total     8      481.67           191.93           673.60

Individual virtual processors:
   vp        pid         class    usercpu          syscpu           total
    1        1776         cpu     165.18           40.50           205.68
    2        1777         adm       0.10            0.45             0.55
    3        1778         cpu     157.83           98.68           256.52
    4        1779         cpu     156.75           51.23           207.98
    5        1780         lio       0.27            0.22             0.48
    6        1781         pio       0.42            0.10             0.52
    7        1782         aio       0.83            0.23             1.07
    8        1783         msc       0.28            0.52             0.80
                          tot     481.67          191.93           673.60
```

Use the **onstat -g ioq** option to determine whether you need to allocate additional AIO virtual processors. The command **onstat -g ioq** displays the length of the I/O queues under the column **len**. You can also see the maximum queue length (since the database server started) in the **maxlen** column. Each chunk serviced by the AIO virtual processors has one line in the **onstat -g ioq** output, identified by the **gfd** queue name. You can correlate the line in **onstat -g ioq** with the actual chunk because the chunks are in the same order as in the **onstat -d** output. For example, in the **onstat -g ioq** output in Figure 12-3, there are two **gfd** queues. The first **gfd** queue holds requests for **root_chunk** because it corresponds to the first chunk shown in the **onstat -d** output in Figure 12-3. Likewise, the second **gfd** queue holds requests for **chunk1** because it corresponds to the second chunk in the **onstat -d** output.

If the database server has a mixture of raw devices and cooked files, the **gfd** queues correspond only to the cooked files in **onstat -d** output.

```
onstat -g ioq

AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
  adt   0      0      0        0        0        0        0
  msc   0      0      1       12        0        0        0
  aio   0      0      4       89       68        0        0
  pio   0      0      1        1        0        1        0
  lio   0      0      1       17        0       17        0
  kio   0      0      0        0        0        0        0
  gfd   3      0      3      254      242       12        0
  gfd   4      0     17      614      261      353        0

smoke% onstat -d
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
a1de1d8  1        1        1        1        N        informix rootdbs
a1df550  2        1        2        1        N        informix space1
 2 active, 2047 maximum
Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
a1de320  1   1   0        75000    66447             PO-   /ix/root_chunk
a1df698  2   2   0        500      447               PO-   /ix//chunk1
 2 active, 2047 maximum
```

Figure 12-3
onstat -g ioq and onstat -d Output

If the length of the I/O queue is growing, I/O requests are accumulating faster than the AIO virtual processors can process them. If the length of the I/O queue continues to show that I/O requests are accumulating, consider adding AIO virtual processors.

Use the **onstat -g rea** option to monitor the number of threads in the ready queue. If the number of threads in the ready queue is growing for a class of virtual processors (for example, the CPU class), you might have to add more of those virtual processors to your configuration. Figure 12-4 shows **onstat -g rea** output.

```
Ready threads:
tid     tcb     rstcb   prty    status                  vp-class   name

6       536a38  406464  4       ready                     3cpu     main_loop()
28      60cfe8  40a124  4       ready                     1cpu     onmode_mon
33      672a20  409dc4  2       ready                     3cpu     sqlexec
```

Figure 12-4
onstat -g rea Output

Monitoring Virtual Processors with SMI Tables

Query the **sysvpprof** table to obtain information on the virtual processors that are currently running. This table contains the following columns.

Column	Description
vpid	Virtual-processor ID number
class	Virtual-processor class
usercpu	Minutes of user CPU consumed
syscpu	Minutes of system CPU consumed

Chapter

13 Shared Memory

In This Chapter

This chapter describes the content of database server shared memory, the factors that determine the sizes of shared-memory areas, and how data moves into and out of shared memory. For information on how to change the database server configuration parameters that determine shared memory allocations, refer to Chapter 14, "Managing Shared Memory."

Each coserver has its own shared memory. The shared memory as discussed in this chapter is for an individual coserver.

Shared Memory

Shared memory is an operating-system feature that allows the database server threads and processes to share data by sharing access to pools of memory. The database server uses shared memory for the following purposes:

- To reduce memory usage and disk I/O
- To perform high-speed communication between processes

Shared memory enables the database server to reduce overall memory uses because the participating processes, in this case, virtual processors, do not need to maintain private copies of the data that is in shared memory.

Shared memory reduces disk I/O because buffers, which are managed as a common pool, are flushed on a database server-wide basis instead of a per-process basis. Furthermore, a virtual processor can often avoid reading data from disk because the data is already in shared memory as a result of an earlier read operation. The reduction in disk I/O reduces execution time.

Shared memory provides the fastest method of interprocess communication because it processes read and write messages at the speed of memory transfers.

Shared-Memory Use

The database server uses shared memory for the following purposes:

- To enable virtual processors and utilities to share data
- To provide a fast communications channel for local client applications that use IPC communication

Figure 13-1 illustrates the shared-memory scheme.

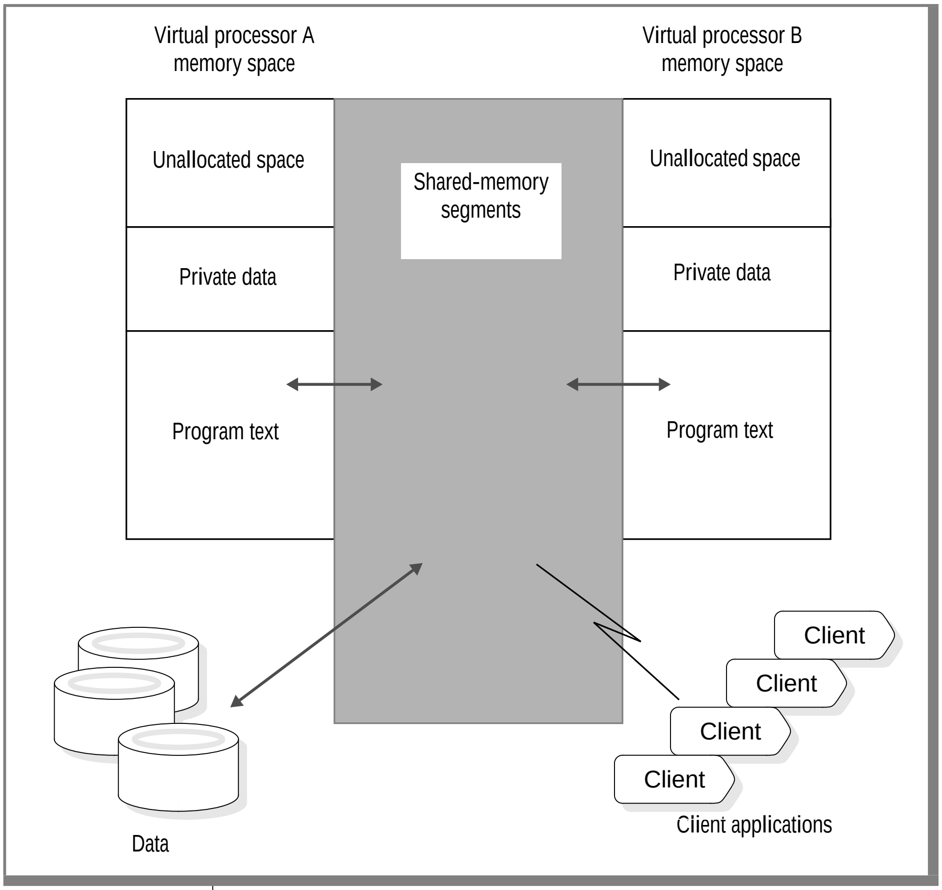


Figure 13-1
How the Database Server Uses Shared Memory

Shared-Memory Allocation

The database server creates the following portions of shared memory:

- The *resident* portion
- The *virtual* portion
- The *IPC communications* or message portion

Each portion of shared memory consumes one or more operating-system segments. When the database server initializes shared memory, it allocates at least two operating-system segments, one for the resident portion and one for the virtual portion. It might allocate more segments if the maximum segment size is not large enough.

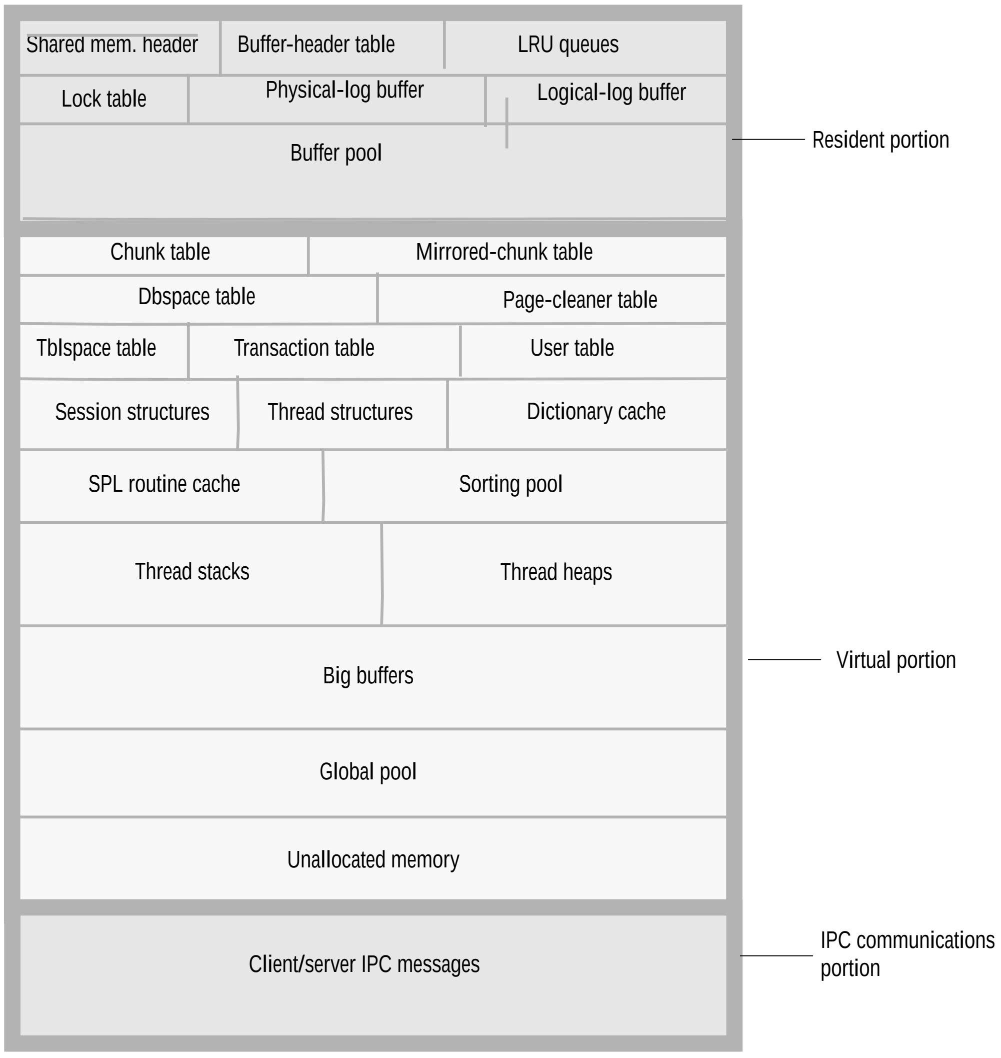
Depending upon operating system settings for shared memory, the database server may allocate one segment for both the resident portion and virtual portion.

If the **sqlhosts** file specifies shared-memory communications, the database server allocates memory for the communications portion.

The database server adds operating-system segments as needed to the virtual portions of shared memory. Figure 13-2 shows the contents of each portion of shared memory.

All database server virtual processors have access to the same shared-memory segments. Each virtual processor manages its work by maintaining its own set of pointers to shared-memory resources such as buffers, locks, and latches. Virtual processors attach to shared memory when you take the database server from off-line mode to quiescent mode or from off-line mode directly to on-line mode. The database server uses locks and latches to manage concurrent access to shared-memory resources by multiple threads. For more information about modes, refer to Chapter 9, "Managing Database Server Operating Modes."

Figure 13-2
Contents of Database Server Shared Memory



Shared-Memory Size

Each portion of the database server shared memory consists of one or more operating-system segments of memory, each one divided into a series of blocks that are 8 kilobytes in size and managed by a bit map.

The header-line output by the **onstat** utility contains the size of the database server shared memory, expressed in kilobytes. For information on how to use **onstat**, refer to the utilities chapter in the *Administrator's Reference*. You can also use the **-g seg** option of **onstat** to monitor how much memory the database server allocates for each portion of shared memory.

You can set the SHMTOTAL parameter in the ONCONFIG file to limit the amount of memory overhead that the database server can place on your computer or node. The SHMTOTAL parameter specifies the total amount of shared memory that the database server can use for all memory allocations. However, certain operations might fail if the database server needs more memory than the amount set in SHMTOTAL. If this condition occurs, the database server displays the following message in the message log:

```
size of resident + virtual segments x + y > z
    total allowed by configuration parameter SHMTOTAL
```

In addition, the database server returns an error message to the application that initiated the offending operation. For example, if the database server needs more memory than you specify in SHMTOTAL while it tries to perform an operation such as an index build or a hash join, it returns an error message to the application that is similar to one of the following:

```
-567    Cannot write sorted rows.
-116    ISAM error: cannot allocate memory.
```

After the database server sends these messages, it rolls back any partial results performed by the offending query.

Internal operations, such as page-cleaner or checkpoint activity, can also cause the database server to exceed the SHMTOTAL ceiling. When this situation occurs, the database server sends a message to the message log. For example, suppose that the database server attempts and fails to allocate additional memory for page-cleaner activity. As a consequence, the database server sends a message to the message log that is similar to the following:

```
17:19:13   Assert Failed: WARNING! No memory available for page cleaners
17:19:13   Who: Thread(11, flush_sub(0), 9a8444, 1)
17:19:13   Results: Database server may be unable to complete a checkpoint
17:19:13   Action: Make more virtual memory available to database server
17:19:13   See Also: /tmp/af.c4
```

After the database server informs you about the failure to allocate additional memory, it rolls back the transactions that caused it to exceed the SHMTOTAL limit. Immediately after the rollback, operations no longer fail from lack of memory, and the database server continues to process transactions as usual.

Action to Take If SHMTOTAL Is Exceeded

When the database server needs more memory than SHMTOTAL allows, a transient condition occurs, perhaps caused by a burst of activity that exceeds the normal processing load. Only the operation that caused the database server to run out of memory temporarily should fail. Other operations continue to be processed in a normal fashion.

If messages indicate on a regular basis that the database server needs more memory than SHMTOTAL allows, you have not configured the database server correctly. Lowering the value of BUFFERS or DS_TOTAL_MEMORY is one possible solution, and increasing the value of SHMTOTAL is another.

Processes That Attach to Shared Memory

The following processes attach to the database server shared memory:

- Client-application processes that communicate with the database server through the shared-memory communications portion (ipcshm)
- Database server virtual processors
- Database server utilities

The following sections describe how each type of process attaches to the database server shared memory.

How a Client Attaches to the Communications Portion

Client-application processes that communicate with the database server through shared memory (**nettype** `ipcshm`) attach transparently to the communications portion of shared memory. System-library functions that are automatically compiled into the application enable it to attach to the communications portion of shared memory. For information on specifying a shared-memory connection, see Chapter 6, "Client / Server Communications," and "Network Virtual Processors" on page 11-26.

If the **INFORMIXSHMBASE** environment variable is not set, the client application attaches to the communications portion at an address that is platform specific. If the client application attaches to other shared-memory segments (not database server shared memory), the user can set the **INFORMIXSHMBASE** environment variable to specify the address at which to attach the database server shared-memory communications segments. When you specify the address at which to address the shared-memory communications segments, you can prevent the database server from colliding with the other shared-memory segments that your application uses. For information on how to set the **INFORMIXSHMBASE** environment variable, refer to the *Informix Guide to SQL: Reference*.

How Utilities Attach to Shared Memory

The database server utilities such as **onstat** and **onmode** attach to shared memory through the **$INFORMIXDIR/etc/.infos.*servername*** file.

The variable ***servername*** is the value of the DBSERVERNAME parameter in the ONCONFIG file. The utilities obtain the ***servername*** portion of the filename from the **INFORMIXSERVER** environment variable.

The **oninit** process reads the ONCONFIG file and creates the file **.infos.*servername*** when it starts the database server. The file is removed when the database server terminates.

How Virtual Processors Attach to Shared Memory

The database server virtual processors attach to shared memory during initialization. During this process, the database server must satisfy the following two requirements:

- Ensure that all virtual processors can locate and access the same shared-memory segments
- Ensure that the shared-memory segments reside in physical memory locations that are different than the shared-memory segments assigned to other instances of the database server, if any, on the same computer

The database server uses two configuration parameters, SERVERNUM and SHMBASE, to meet these requirements.

When a virtual processor attaches to shared memory, it performs the following major steps:

1. Accesses the SERVERNUM parameter from the ONCONFIG file
2. Uses SERVERNUM to calculate a shared-memory key value
3. Requests a shared-memory segment using the shared-memory key value

   The operating system returns the shared-memory identifier for the first shared-memory segment.
4. Directs the operating system to attach the first shared-memory segment to its process space at SHMBASE
5. Attaches additional shared-memory segments, if required, to be contiguous with the first segment

The following sections describe how the database server uses the values of the SERVERNUM and SHMBASE configuration parameters in the process of attaching shared-memory segments.

Defining a Unique Key Value

The database server uses the ONCONFIG parameter SERVERNUM to calculate a unique key value for its shared-memory segments. All virtual processors within a single database server instance share the same key value. When each virtual processor attaches to shared memory, it calculates the key value as follows:

```
(SERVERNUM * 65536) + shmkey
```

The value of *shmkey* is set internally and cannot be changed by the user. (The *shmkey* value is `52564801` in hexadecimal representation or `1,381,386,241` in decimal.) The value (`SERVERNUM * 65,536`) is the same as multiplying SERVERNUM by hexadecimal 10,000.

When more than one database server instance exists on a single computer, the difference in the key values for any two instances is the difference between the two SERVERNUM values, multiplied by 65,536.

When a virtual processor requests that the operating system attach the first shared-memory segment, it supplies the unique key value to identify the segment. In return, the operating system passes back a *shared-memory segment identifier* associated with the key value. Using this identifier, the virtual processor requests that the operating system attach the segment of shared memory to the virtual-processor address space.

Specifying Where to Attach the First Shared-Memory Segment

The SHMBASE parameter in the ONCONFIG file specifies the virtual address where each virtual processor attaches the first, or base, shared-memory segment. Each virtual processor attaches to the first shared-memory segment at the same virtual address. This situation enables all virtual processors within the same database server instance to reference the same locations in shared memory without needing to calculate shared-memory addresses. All shared-memory addresses for an instance of the database server are relative to SHMBASE.

Warning: *Informix recommends that you not attempt to change the value of SHMBASE.*

The value of SHMBASE is sensitive for the following reasons:

- The specific value of SHMBASE is often computer dependent. It is not an arbitrary number. Informix selects a value for SHMBASE that keeps the shared-memory segments safe when the virtual processor dynamically acquires additional memory space.
- Different operating systems accommodate additional memory at different virtual addresses. Some architectures extend the highest virtual address of the virtual-processor data segment to accommodate the next segment. In this case, the data segment might grow into the shared-memory segment.
- Some platforms require the user to specify a SHMBASE parameter of virtual address zero. The zero address informs the UNIX kernel that the kernel should pick the best address at which to attach the shared-memory segments. However, not all platforms support this option. Moreover, on some systems, the selection that the kernel makes might not be the best selection.

Attaching Additional Shared-Memory Segments

Each virtual processor must attach to the total amount of shared memory that the database server has acquired. After a virtual processor attaches each shared-memory segment, it calculates how much shared memory it has attached and how much remains. The database server facilitates this process by writing a shared-memory header to the first shared-memory segment. Sixteen bytes into the header, a virtual processor can obtain the following data:

- The total size of shared memory for this database server
- The size of each shared-memory segment

To attach additional shared-memory segments, a virtual processor requests them from the operating system in much the same way that it requested the first segment. For the additional segments, however, the virtual processor adds 1 to the previous value of *shmkey*. The virtual processor directs the operating system to attach the segment at the address that results from the following calculation:

```
SHMBASE + (seg_size x number of attached segments)
```

The virtual processor repeats this process until it has acquired the total amount of shared memory.

Given the initial key value of (`SERVERNUM * 65536`) + *shmkey*, the database server can request up to 65,536 shared-memory segments before it could request a shared-memory key value used by another database server instance on the same computer.

Defining the Shared-Memory Lower-Boundary Address

If your operating system uses a parameter to define the lower boundary address for shared memory, and the parameter is set incorrectly, it can prevent the shared-memory segments from being attached contiguously.

Figure 13-3 illustrates the problem. If the lower-boundary address is less than the ending address of the previous segment plus the size of the current segment, the operating system attaches the current segment at a point beyond the end of the previous segment. This action creates a gap between the two segments. Because shared memory must be attached to a virtual processor so that it looks like contiguous memory, this gap creates problems. The database server receives errors when this situation occurs. To correct the problem, check the operating-system kernel parameter that specifies the lower-boundary address or reconfigure the kernel to allow larger shared-memory segments. For a description of the operating-system kernel parameter, refer to "Shared-Memory Lower-Boundary Address" on page 14-6.

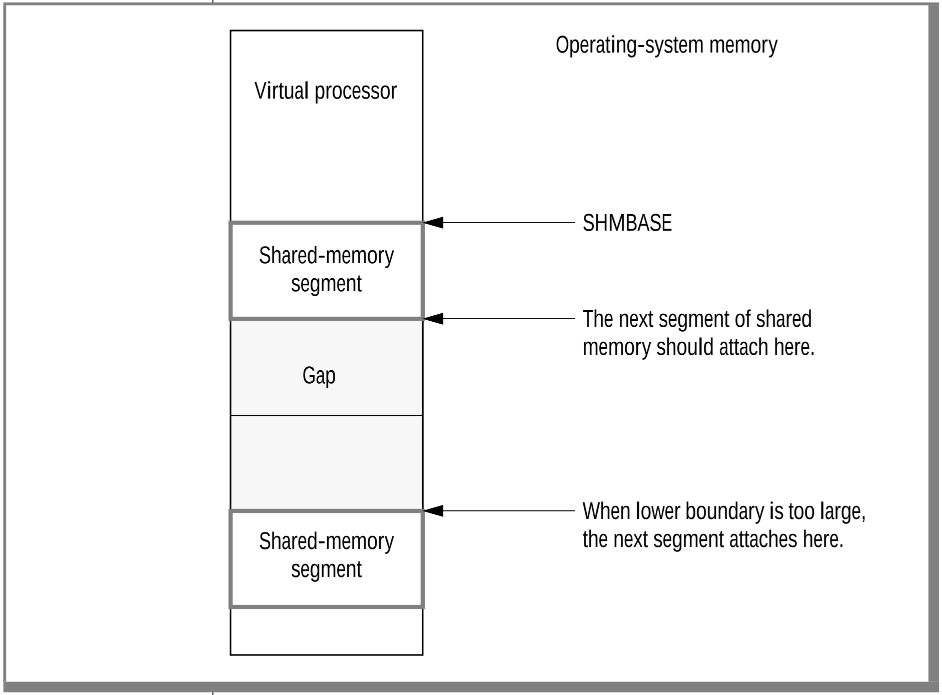

Operating-system memory
Virtual processor
SHMBASE
Shared-memory segment
The next segment of shared memory should attach here.
Gap
When lower boundary is too large, the next segment attaches here.
Shared-memory segment

Figure 13-3
Shared-Memory Lower-Boundary Address Overview

Resident Shared-Memory Segments

The operating system, as it switches between the processes running on the system, normally swaps the contents of portions of memory to disk. When a portion of memory is designated as *resident*, however, it is not swapped to disk. Keeping frequently accessed data resident in memory improves performance because it reduces the number of disk I/O operations that would otherwise be required to access that data.

The database server requests that the operating system keep the virtual portions in physical memory when the following two conditions exist:

- The operating system supports shared-memory residency.
- The RESIDENT parameter in the ONCONFIG file is set to `-1` or a value that is greater than 0.

Warning: *You must consider the use of shared memory by all applications when you consider whether to set the RESIDENT parameter to `-1`. Locking all shared memory for the use of the Informix database server can adversely affect the performance of other applications, if any, on the same computer.*

For more information on the RESIDENT configuration parameter, refer to the *Administrator's Reference*.

Resident Portion of Shared Memory

The resident portion of the database server shared memory stores the following data structures that do not change in size while the database server is running:

- Shared-memory header
- Buffer pool
- Logical-log buffer
- Physical-log buffer
- Lock table

Shared-Memory Header

The shared-memory header contains a description of all other structures in shared memory, including internal tables and the buffer pool.

The shared-memory header also contains pointers to the locations of these structures. When a virtual processor first attaches to shared memory, it reads address information in the shared-memory header for directions to all other structures.

The size of the shared-memory header is about one kilobyte, but the size varies depending on the computer platform. You cannot tune the size of the header.

Shared-Memory Buffer Pool

The buffer pool in the resident portion of shared memory contains buffers that store dbspace pages read from disk. The pool of buffers comprises the largest allocation of the resident portion of shared memory.

Figure 13-4 illustrates the shared-memory header and the buffer pool.

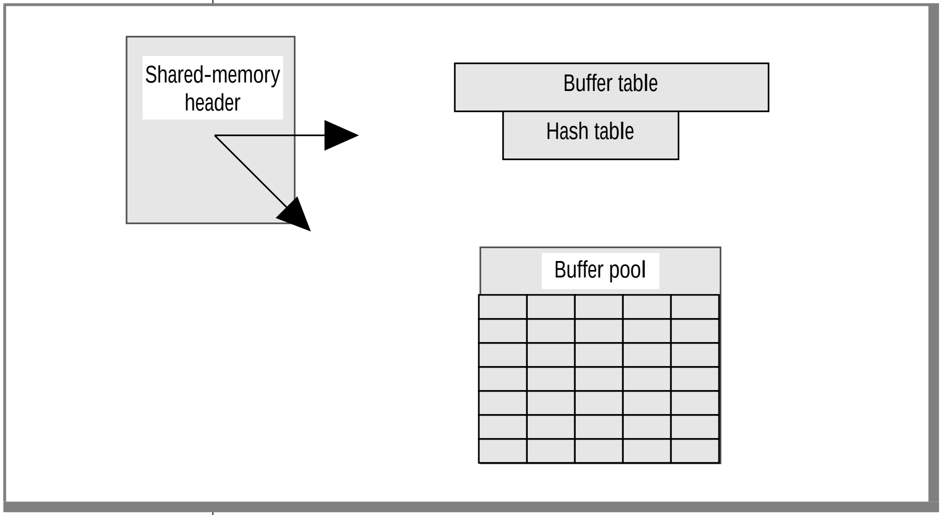


Figure 13-4
Shared-Memory Buffer Pool

You specify the number of buffers in the buffer pool with the BUFFERS parameter in the ONCONFIG file. BUFFERS defaults to 1000 buffers. To allocate the proper number of buffers, start with at least four buffers per user. For more than 500 users, the minimum requirement is 2000 buffers. Too few buffers can severely impact performance, so you must monitor the database server and tune the value of BUFFERS to determine an acceptable value. For more information on tuning the number of buffers, refer to your *Performance Guide*.

For more information on setting the BUFFERS configuration parameter, refer to the *Administrator's Reference*.

The status of the buffers is tracked through the buffer table. Within shared memory, buffers are organized into LRU buffer queues. Buffer acquisition is managed through the use of latches, called *mutexes*, and lock-access information.

For a description of how LRU queues work, refer to "LRU Queues" on page 13-36. For a description of mutexes, refer to "Mutexes" on page 11-17.

Buffer Overflow to the Virtual Portion

Because the maximum number of buffers in 64-bit addressing can be as large as 2^{31}-1, the resident portion of shared memory might not be able to hold all of the buffers in a large buffer pool. In this case, the virtual portion of database server shared memory might hold some of the buffers.

Buffer Size

Each buffer is the size of one database server page. In general, the database server performs I/O in full-page units, the size of a buffer. The exception is I/O performed from big buffers. See "Big Buffers" on page 13-30.

The **-b** option of the **onstat** utility displays information about the buffers. For information on **onstat**, refer to the utilities chapter in the *Administrator's Reference*.

Configurable Page Size

You can use the PAGESIZE parameter to specify a database server page size of 2048, 4096, or 8192 bytes. The **-b** option of the **onstat** utility also displays the database server page size.

If your workload is mostly DSS queries, a larger page size gives you better performance. If your workload is mostly OLTP, a smaller page size gives you better performance. For more information about the PAGESIZE parameter, see the *Administrator's Reference*.

The page size affects the calculations for the buffer pool, logical-log buffer, physical-log buffer, physical-log size, and the maximum logical-log size. For more information, see the chapter on effects of configuration on memory use in your *Performance Guide*.

You can use the **-p** and **-P** options of the **onstat** utility to determine the rate of buffer pool usage and to optimize the use of memory resident tables, respectively. The **onstat -t** option enables you to determine which tblspaces are resident. For information on **onstat**, refer to the utilities chapter in the *Administrator's Reference*.

Logical-Log Buffer

The database server uses the logical log to store a record of changes to the database server data since the last dbspace backup. The logical log stores records that represent logical units of work for the database server. The logical log contains the following five types of log records, in addition to many others:

- SQL data definition statements for all databases
- SQL data manipulation statements for databases that were created with logging
- Record of a change to the logging status of a database
- Record of a full or fuzzy checkpoint
- Record of a change to the configuration

The database server uses only one of the logical-log buffers at a time. This buffer is the current logical-log buffer. Before the database server flushes the current logical-log buffer to disk, it makes the second logical-log buffer the current one so that it can continue writing while the first buffer is flushed. If the second logical-log buffer fills before the first one finishes flushing, the third logical-log buffer becomes the current one. This process is illustrated in Figure 13-5.

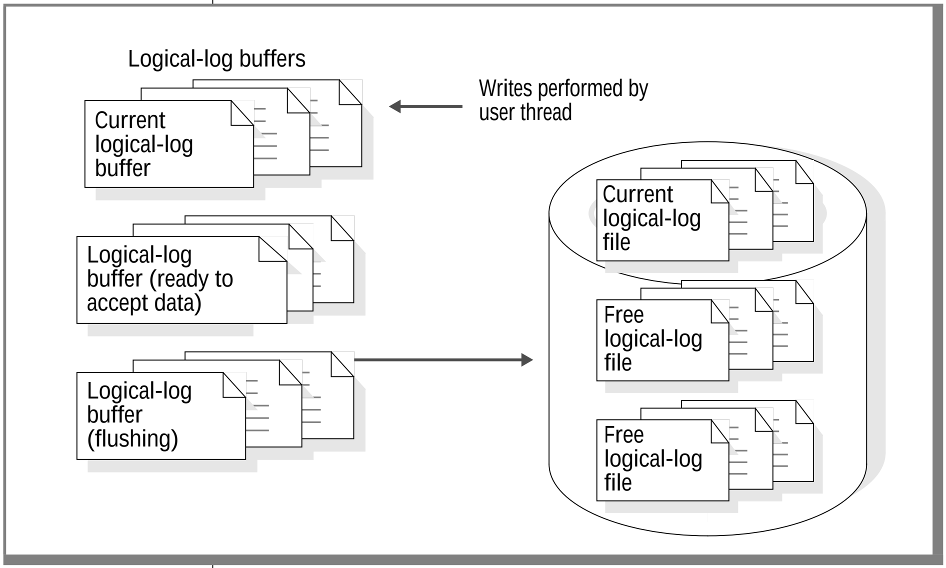

Logical-log buffers
Current logical-log buffer
Logical-log buffer (ready to accept data)
Logical-log buffer (flushing)
Writes performed by user thread
Current logical-log file
Free logical-log file
Free logical-log file

Figure 13-5
The Logical-Log Buffer and Its Relation to the Logical-Log Files on Disk

For a description of how the database server flushes the logical-log buffer, refer to "Flushing the Logical-Log Buffer" on page 13-52.

The LOGBUFF parameter in the ONCONFIG file specifies the size of the logical-log buffers. Small buffers can create problems if you store records larger than the size of the buffers (for example, TEXT or BYTE data in dbspaces). For the possible values that you can assign to this configuration parameter, refer to the *Administrator's Reference*.

For information on the impact of TEXT and BYTE data on shared memory buffers, refer to "Buffering Simple-Large-Object Data Types" on page 13-55.

Physical-Log Buffer

The database server uses the physical-log buffer to hold before-images of some of the modified dbspace pages. The before-images in the physical log and the logical-log records enable the database server to restore consistency to its databases after a system failure.

The physical-log buffer is actually two buffers. Double buffering permits the database server processes to write to the active physical-log buffer while the other buffer is being flushed to the physical log on disk. For a description of how the database server flushes the physical-log buffer, refer to "Flushing the Physical-Log Buffer" on page 13-46. For information on monitoring the physical-log file, refer to "Monitoring Physical and Logical Logging Activity" on page 23-5.

The PHYSBUFF parameter in the ONCONFIG file specifies the size of the physical-log buffers. A write to the physical-log buffer writes exactly one page. If the specified size of the physical-log buffer is not evenly divisible by the page size, the database server rounds the size down to the nearest value that is evenly divisible by the page size. Although some operations require the buffer to be flushed sooner, in general the database server flushes the buffer to the physical-log file on disk when the buffer fills. Thus, the size of the buffer determines how frequently the database server needs to flush it to disk. For more information on this configuration parameter, refer to the *Administrator's Reference*.

Lock Table

A lock is created when a user thread writes an entry in the lock table. The lock table is the pool of available locks. Each entry is one lock. A single transaction can own multiple locks. For an explanation of locking and the SQL statements associated with locking, refer to the *Informix Guide to SQL: Tutorial*. For information on performance considerations for locking, refer to your *Performance Guide*

The following information, which is stored in the lock table, describes the lock:

- The address of the transaction that owns the lock
- The type of lock (exclusive, update, shared, or intent)
- The page and rowid that is locked
- The tblspace where the lock is placed

To specify the number of entries in the lock table, set the LOCKS configuration parameter. However, the lock table grows dynamically if more locks are needed. For information on specifying the number of locks available to sessions, refer to the chapter on configuration parameters in the *Administrator's Reference*. For information on how to estimate the number of locks, refer to the chapter on configuration effects on memory utilization in your *Performance Guide*.

For information on how the lock table grows dynamically and monitoring locks, refer to the chapter on locking in your *Performance Guide*.

Virtual Portion of Shared Memory

The virtual portion of shared memory is expandable by the database server and can be paged out to disk by the operating system. As the database server executes, it automatically attaches additional operating-system segments, as needed, to the virtual portion.

Management of the Virtual Portion of Shared Memory

The database server uses memory *pools* to track memory allocations that are similar in type and size. Keeping related memory allocations in a pool helps to reduce memory fragmentation. It also enables the database server to free a large allocation of memory at one time, as opposed to freeing each piece that makes up the pool.

All sessions have one or more memory pools. When the database server needs memory, it looks first in the specified pool. If insufficient memory is available in a pool to satisfy a request, the database server adds memory from the system pool. If the database server cannot find enough memory in the system pool, it dynamically allocates more segments to the virtual portion.

The database server allocates virtual shared memory for each of its subsystems (session pools, stacks, heaps, control blocks, system catalog, SPL routine caches, SQL statement cache, sort pools, and message buffers) from pools that track free space through a linked list. When the database server allocates a portion of memory, it first searches the pool free-list for a *fragment* of sufficient size. If it finds none, it brings new blocks into the pool from the virtual portion. When memory is freed, it goes back to the pool as a free fragment and remains there until the pool is destroyed. When the database server starts a session for a client application, for example, it allocates memory for the session pool. When the session terminates, the database server returns the allocated memory as free fragments.

Size of the Virtual Portion of Shared Memory

To specify the initial size of the virtual shared-memory portion, set the SHMVIRTSIZE parameter in the ONCONFIG file. To specify the size of segments that are later added to the virtual portion of shared memory, set the SHMADD parameter in the ONCONFIG file.

For more information on determining the size of virtual shared memory, refer to "Adding a Segment to the Virtual Portion of Shared Memory" on page 14-13 and to the SHMVIRTSIZE and SHMADD configuration parameters in the *Administrator's Reference*.

Components of the Virtual Portion of Shared Memory

The virtual portion of shared memory stores the following data:

- Internal tables
- Big buffers
- Session data
- Thread data (stacks and heaps)
- Dictionary cache
- SPL routine cache
- Sorting pool
- Global pool

Shared-Memory Internal Tables

The database server shared memory contains seven internal tables that track shared-memory resources. The shared-memory internal tables are as follows:

- Buffer table
- Chunk table
- Dbspace table
- Page-cleaner table
- Tblspace table
- Transaction table
- User table

Buffer Table

The buffer table tracks the addresses and status of the individual buffers in the shared-memory pool. When a buffer is used, it contains an image of a data or index page from disk. For more information on the purpose and content of a disk page, refer to "Pages" on page 15-10.

Each buffer in the buffer table contains the following control information, which is needed for buffer management:

- Buffer status

  Buffer status is described as empty, unmodified, or modified. An unmodified buffer contains data, but the data can be overwritten. A modified, or dirty buffer, contains data that must be written to disk before it can be overwritten.
- Current lock-access level

  Buffers receive lock-access levels depending on the type of operation that the user thread is executing. The database server supports two buffer lock-access levels: shared and exclusive.
- Threads waiting for the buffer

  Each buffer header maintains a list of the threads that are waiting for the buffer and the lock-access level that each waiting thread requires.

Each database server buffer has one entry in the buffer table.

For information on the database server buffers, refer to "Resident Portion of Shared Memory" on page 13-19. For information on how to monitor the buffers, refer to "Monitoring Buffers" on page 14-15.

The database server determines the number of entries in the buffer-table hash table based on the number of allocated buffers. The maximum number of hash values is the largest power of 2 that is less than the value of BUFFERS.

Chunk Table

The chunk table tracks all chunks in the database server. If mirroring has been enabled, a corresponding mirrored chunk table is also created when shared memory is initialized. The mirrored chunk table tracks all mirrored chunks.

The chunk table in shared memory contains information that enables the database server to locate chunks on disk. This information includes the number of the initial chunk and the number of the next chunk in the dbspace. Flags also describe chunk status: mirror or primary, as well as off-line, on-line, or recovery mode. For information on monitoring chunks, refer to "Monitoring Chunks" on page 16-34.

The maximum number of entries in the chunk table might be limited by the maximum number of file descriptors that your operating system allows per process. You can usually specify the number of file descriptors per process with an operating-system kernel-configuration parameter. For details, consult your operating-system manuals.

Dbspace Table

The dbspace table tracks storage spaces in the database server. The dbspace-table information includes the following information about each dbspace:

- Dbspace number
- Dbspace name and owner
- Dbspace mirror status (mirrored or not)
- Date and time that the dbspace was created

For information on monitoring dbspaces, refer to "Monitoring the Database Server for Disabling I/O Errors" on page 16-32.

Page-Cleaner Table

The page-cleaner table tracks the state and location of each of the page-cleaner threads. The number of page-cleaner threads is specified by the CLEANERS configuration parameter in the ONCONFIG file. For advice on how many page-cleaner threads to specify, refer to the chapter on configuration parameters in the *Administrator's Reference*.

The page-cleaner table always contains 128 entries, regardless of the number of page-cleaner threads specified by the CLEANERS parameter in the ONCONFIG file.

For information on monitoring the activity of page-cleaner threads, refer to the **onstat -F** option in the utilities chapter of the *Administrator's Reference*.

Tblspace Table

The tblspace table tracks all active tblspaces in a database server instance. An active tblspace is one that is currently in use by a database session. Each active table accounts for one entry in the tblspace table. Active tblspaces include database tables, temporary tables, and internal control tables, such as system catalog tables. Each tblspace table entry includes header information about the tblspace, the tblspace name, and pointers to the tblspace tblspace in dbspaces on disk. (The shared-memory active tblspace table is different from the tblspace tblspace.) For information on monitoring tblspaces, refer to "Monitoring Tblspaces and Extents" on page 16-38.

The database server manages one tblspace table for each dbspace.

Transaction Table

The transaction table tracks all transactions in the database server.

Tracking information derived from the transaction table appears in the **onstat -x** display. For an example of the output that **onstat -x** displays, refer to monitoring transactions in your *Performance Guide*.

The database server automatically increases the number of entries in the transaction table, up to a maximum of 32,767, based on the number of current transactions.

For more information on transactions and the SQL statements that you use with transactions, refer to the *Informix Guide to SQL: Tutorial*, the *Informix Guide to SQL: Reference*, and the *Informix Guide to SQL: Syntax*.

The transaction table also specifically supports the X/Open environment. Support for the X/Open environment requires TP/XA. For a description of a transaction in this environment, refer to the *TP/XA Programmer's Manual*.

User Table

The user table tracks all user threads and system threads. Each client session has one primary thread and zero-to-many secondary threads, depending on the level of parallelism specified. System threads include one to monitor and control checkpoints, one to process **onmode** commands, the B-tree cleaner thread, and one or more page-cleaner threads.

The database server increases the number of entries in the user table as needed. You can monitor user threads with the **onstat -u** command.

Big Buffers

A big buffer is a single buffer that is made up of several pages. The actual number of pages is platform dependent. The database server allocates big buffers to improve performance on large reads and writes.

The database server uses a big buffer whenever it writes to disk multiple pages that are physically contiguous. For example, the database server tries to use a big buffer to perform a series of sequential reads (light scans) or to read into shared memory simple large objects that are stored in a dbspace.

For information on monitoring big buffers with the **onstat** command, refer to the chapter on configuration effects on I/O activity in your *Performance Guide*.

Session Data

When a client application requests a connection to the database server, the database server begins a *session* with the client and creates a data structure for the session in shared memory called the *session-control block*. The session-control block stores the session ID, the user ID, the process ID of the client, the name of the host computer, and various status flags.

The database server allocates memory for session data as needed.

Thread Data

When a client connects to the database server, in addition to starting a session, the database server starts a primary session thread and creates a *thread-control block* for it in shared memory.

The database server also starts internal threads on its own behalf and creates thread-control blocks for them. When the database server switches from running one thread to running another one (a context switch), it saves information about the thread— such as the register contents, program counter (address of the next instruction), and global pointers—in the thread-control block. For more information on the thread-control block and how it is used, refer to "Context Switching" on page 11-12.

The database server allocates memory for thread-control blocks as needed.

Stacks

Each thread in the database server has its own stack area in the virtual portion of shared memory. For a description of how threads use stacks, refer to "Stacks" on page 11-13. For information on how to monitor the size of the stack for a session, refer to monitoring sessions and threads section in your *Performance Guide*.

The size of the stack space for user threads is specified by the STACKSIZE parameter in the ONCONFIG file. The default size of the stack is 32 kilobytes. You can change the size of the stack for all user threads, if necessary, by changing the value of STACKSIZE. For information and a warning on setting the size of the stack, refer to STACKSIZE in the chapter on configuration parameters in the *Administrator's Reference*.

To alter the size of the stack for the primary thread of a specific session, set the **INFORMIXSTACKSIZE** environment variable. The value of **INFORMIXSTACKSIZE** overrides the value of STACKSIZE for a particular user. For information on how to override the stack size for a particular user, refer to the description of the **INFORMIXSTACKSIZE** environment variable in the *Informix Guide to SQL: Reference*.

To more safely alter the size of stack space, use the **INFORMIXSTACKSIZE** environment variable rather than alter the configuration parameter STACKSIZE. The **INFORMIXSTACKSIZE** environment variable affects the stack space for only one user, and it is less likely to affect new client applications that initially were not measured.

Heaps

Each thread has a heap to hold data structures that it creates while it is running. A heap is dynamically allocated when the thread is created. The size of the thread heap is not configurable.

Dictionary Cache

When a session executes an SQL statement that requires access to a system catalog table, the database server reads the system catalog tables and stores them in structures that it can access more efficiently. These structures are created in the virtual portion of shared memory for use by all sessions. These structures constitute the dictionary cache.

You can configure the size of the dictionary cache with the DD_HASHSIZE and DD_HASHMAX configuration parameters. For more information about these parameters, refer to the chapter on configuration effects on memory in your *Performance Guide*.

Sorting Memory

The following database operations can use large amounts of the virtual portion of shared memory to sort data:

- Decision-support queries that involve joins, groups, aggregates and sort operations
- Index builds
- UPDATE STATISTICS statement in SQL

The amount of virtual shared memory that the database server allocates for a sort depends on the number of rows to be sorted and the size of the row, along with other factors.

For information on parallel sorts, refer to your *Performance Guide*.

SPL Routine Cache

When a session needs to access an SPL routine for the first time, the database server reads the definition from the system catalog tables and stores the definition in a cache. The database server converts the SPL routine to executable format and stores the routine in the cache, where it can be accessed by any session.

You can configure the size of the SPL routine cache with the PC_HASHSIZE and PC_POOLSIZE configuration parameters. For information about changing the default size of the SPL routine cache, refer to the chapter on queries and the query optimizer in your *Performance Guide*.

Global Pool

The global pool stores structures that are global to the database server. For example, the global pool contains the message queues where poll threads for network communications deposit messages from clients. The **sqlexec** threads pick up the messages from the global pool and process them.

Communications Portion of Shared Memory

The database server allocates memory for the IPC communication portion of shared memory if you configure at least one of your connections as an IPC shared-memory connection. The database server performs this allocation when you initialize shared memory. The communications portion contains the message buffers for local client applications that use shared memory to communicate with the database server.

The size of the communications portion of shared memory equals approximately 12 kilobytes multiplied by the expected number of connections needed for shared-memory communications (**nettype** `ipcshm`). If **nettype** `ipcshm` is not present, the expected number of connections defaults to `50`. For information about how a client attaches to the communications portion of shared memory, refer to "How a Client Attaches to the Communications Portion" on page 13-12.

Concurrency Control

The database server threads that run on the same virtual processor, and on separate virtual processors, share access to resources in shared memory. When a thread writes to shared memory, it uses mechanisms called *mutexes* and *locks* to prevent other threads from simultaneously writing to the same area. A mutex gives a thread the right to access a shared-memory resource. A lock prevents other threads from writing to a buffer until the thread that placed the lock is finished with the buffer and releases the lock.

Shared-Memory Mutexes

The database server uses *mutexes* to coordinate threads as they attempt to modify data in shared memory. Every modifiable shared-memory resource is associated with a mutex. Before a thread can modify a shared-memory resource, it must first acquire the mutex associated with that resource. After the thread acquires the mutex, it can modify the resource. When the modification is complete, the thread releases the mutex.

If a thread tries to obtain a mutex and finds that it is held by another thread, the incoming thread must wait for the mutex to be released.

For example, two threads can attempt to access the same slot in the chunk table, but only one can acquire the mutex associated with the chunk table. Only the thread that holds the mutex can write its entry in the chunk table. The second thread must wait for the mutex to be released and then acquire it.

For information on monitoring mutexes (which are also referred to as latches in the output from the monitoring tools), refer to "Monitoring Latches" on page 14-22.

Shared-Memory Buffer Locks

A primary benefit of shared memory is the ability of database server threads to share access to disk pages stored in the shared-memory buffer pool. The database server maintains thread isolation while it achieves this increased concurrency through a strategy for locking the data buffers.

Types of Buffer Locks

The database server uses two types of locks to manage access to shared-memory buffers:

- Share locks
- Exclusive locks

Each of these lock types enforces the required level of thread isolation during execution.

Share Lock

A buffer is in share mode, or has a share lock, if multiple threads have access to the buffer to read the data but none intends to modify the data.

Exclusive Lock

A buffer is in exclusive mode, or has an exclusive lock, if a thread demands exclusive access to the buffer. All other thread requests that access the buffer are placed in the wait queue. When the executing thread is ready to release the exclusive lock, it wakes the next thread in the wait queue.

Database Server Thread Access to Shared Buffers

Database server threads access shared buffers through a system of queues, using mutexes and locks to synchronize access and protect data.

LRU Queues

Each buffer in the buffer pool is tracked through several linked lists of pointers to the buffer table. A buffer holds data for the purpose of caching. These linked lists are the least-recently used (LRU) queues.

The LRUS parameter in the ONCONFIG file specifies the number of LRU queues to create when database server shared memory is initialized. You can tune the value of LRUS, combined with the LRU_MIN_DIRTY and LRU_MAX_DIRTY parameters, to control how frequently the shared-memory buffers are flushed to disk.

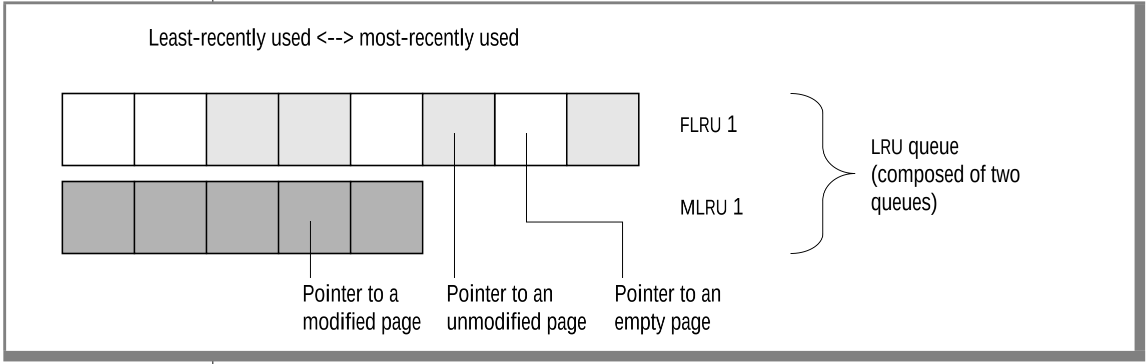
Components of LRU Queue

Each LRU queue is composed of a pair of linked lists, as follows:

- FLRU (free least-recently used) list, which tracks free or unmodified pages in the queue
- MLRU (modified least-recently used) list, which tracks modified pages in the queue

The free or unmodified page list is referred to as the FLRU queue of the queue pair, and the modified page list is referred to as the MLRU queue. The two separate lists eliminate the need to search a queue for a free or unmodified page. Figure 13-6 illustrates the structure of the LRU queues.

Figure 13-6
LRU Queue


Least-recently used <--> most-recently used
FLRU 1
MLRU 1
LRU queue (composed of two queues)
Pointer to a modified page
Pointer to an unmodified page
Pointer to an empty page

Pages in Least-Recently Used Order

When the database server processes a request to read a page from disk, it must decide which page to replace in memory. Rather than select a page randomly, the database server assumes that recently referenced pages are more likely to be referenced in the future than pages that it has not referenced for some time. Thus, rather than replacing a recently accessed page, the database server replaces a least-recently accessed page. By maintaining pages in least-recently to most-recently used order, the database server can easily locate the least-recently used pages in memory.

LRU Queues and Buffer-Pool Management

Before processing begins, all page buffers are empty, and every buffer is represented by an entry in one of the FLRU queues. The buffers are evenly distributed among the FLRU queues. To calculate the number of buffers in each queue, divide the total number of buffers (BUFFERS) by the number of LRU queues (LRUS).

When a user thread needs to acquire a buffer, the database server randomly selects one of the FLRU queues and uses the oldest or *least-recently used* entry in the list. If the least-recently used page can be latched, that page is removed from the queue.

If the FLRU queue is locked, and the end page cannot be latched, the database server randomly selects another FLRU queue.

If a user thread is searching for a specific page in shared memory, it obtains the LRU-queue location of the page from the control information stored in the buffer table.

After an executing thread finishes its work, it releases the buffer. If the page has been modified, the buffer is placed at the *most-recently used* end of an MLRU queue. If the page was read but not modified, the buffer is returned to the FLRU queue at its most-recently used end. For information on how to monitor LRU queues, refer to "Monitoring Buffer-Pool Activity" on page 14-18.

Number of LRU Queues to Configure

Multiple LRU queues have two purposes:

- They reduce user-thread contention for the queues.
- They allow multiple cleaners to flush pages from LRU queues and maintain the percentage of dirty pages at an acceptable level.

Informix recommends initial values for the LRUS configuration parameter based on the number of CPUs that are available on your computer or node. If your computer is a uniprocessor, start by setting LRUS to 4. If your computer is a multiprocessor, use the following formula:

```
LRUS = max(4, (NUMCPUVPS))
```

After you provide an initial value to LRUS, monitor your LRU queues with **onstat -R**. If you find that the percent of dirty LRU queues consistently exceeds the value of the LRU_MAX_DIRTY parameter, increase the value of the LRUS configuration parameter to add more LRU queues.

For example, suppose you set LRU_MAX_DIRTY to 70 and find that your LRU queues are consistently 75 percent dirty. Consider increasing the value of the LRUS configuration parameter. If you increase the number of LRU queues, you shorten the length of the queues, thereby reducing the work of the page cleaners. However, you must allocate a sufficient number of page cleaners with the CLEANERS configuration parameter, as discussed in the following section.

Number of Cleaners to Allocate

In general, Informix recommends that you configure one cleaner for each disk that your applications update frequently. However, you should also consider the length of your LRU queues and frequency of checkpoints, as explained in the following paragraphs.

In addition to insufficient LRU queues, another factor that influences whether page cleaners keep up with the number of pages that require cleaning can occur if you do not have enough page-cleaner threads allocated. The percent of dirty pages might exceed LRU_MAX_DIRTY in some queues because no page cleaners are available to clean the queues. After a while, the page cleaners might be too far behind to catch up, and the buffer pool becomes much more dirty than the percent that you specified in LRU_MAX_DIRTY.

For example, suppose that the CLEANERS parameter is set to 8, and you increase the number of LRU queues from 8 to 12. You can expect little in the way of a performance gain because the 8 cleaners must now share the work of cleaning an additional 4 queues. If you increase the number of CLEANERS to 12, each of the now-shortened queues can be more efficiently cleaned by a single cleaner.

Setting CLEANERS too low can cause performance to suffer whenever a checkpoint occurs because page cleaners must flush all modified pages to disk during checkpoints. If you do not configure a sufficient number of page cleaners, checkpoints take longer, causing overall performance to suffer.

Number of Pages Added to the MLRU Queues

Periodically, the page-cleaner threads flush the modified buffers in an MLRU queue to disk. To specify the point at which cleaning begins, use the LRU_MAX_DIRTY configuration parameter.

By specifying when page cleaning begins, the LRU_MAX_DIRTY configuration parameter limits the number of page buffers that can be appended to an MLRU queue. The initial setting of LRU_MAX_DIRTY is `60`, so page cleaning begins when 60 percent of the buffers managed by a queue are modified.

You can set LRU_MAX_DIRTY to values less than 1 for example, `.1`).

In practice, page cleaning begins under several conditions, only one of which is when an MLRU queue reaches the value of LRU_MAX_DIRTY. For more information on how the database server performs buffer-pool flushing, refer to "Flushing Data to Disk" on page 13-45.

Figure 13-7 shows how the value of LRU_MAX_DIRTY is applied to an LRU queue to specify when page cleaning begins and thereby limit the number of buffers in an MLRU queue.

```
BUFFERS specified as 8000
LRUS specified as 8
LRU_MAX_DIRTY specified as 60

Page cleaning begins when the number of buffers in the MLRU
    queue is equal to LRU_MAX_DIRTY.

Buffers per LRU queue = (8000/8) = 1000

Max buffers in MLRU queue and point at which page cleaning
    begins: 1000 x 0.60 = 600
```

Figure 13-7
How LRU_MAX_DIRTY Initiates Page Cleaning to Limit the Size of the MLRU Queue

End of MLRU Cleaning

You can also specify the point at which MLRU cleaning can end. The LRU_MIN_DIRTY configuration parameter specifies the acceptable percent of buffers in an MLRU queue. The initial setting of LRU_MIN_DIRTY is `50`, meaning that page cleaning is no longer required when 50 percent of the buffers in an LRU queue are modified. In practice, page cleaning can continue beyond this point as directed by the page-cleaner threads.

You can set LRU_MIN_DIRTY to values less than 1, for example, `.1`.

Figure 13-8 shows how the value of LRU_MIN_DIRTY is applied to the LRU queue to specify the acceptable percent of buffers in an MLRU queue and the point at which page cleaning ends.

```
BUFFERS specified as 8000
LRUS specified as 8
LRU_MIN_DIRTY specified as 50

The acceptable number of buffers in the MLRU queue and
    the point at which page cleaning can end is equal
    to LRU_MIN_DIRTY.

Buffers per LRU queue = (8000/8) = 1000

Acceptable number of buffers in MLRU queue and the point
    at which page cleaning can end: 1000 x .050 = 50
```

Figure 13-8
How LRU_MIN_DIRTY Specifies the Point at Which Page Cleaning Can End

For more information on how the database server flushes the buffer pool, refer to "Flushing Data to Disk" on page 13-45.

Configuring the Database Server to Read Ahead

For sequential table or index scans, you can configure the database server to read several pages ahead while the current pages are being processed. A read-ahead enables applications to run faster because they spend less time waiting for disk I/O.

The database server performs a read-ahead whenever it detects the need for it during sequential data or index reads.

The RA_PAGES parameter in the ONCONFIG file specifies the number of data pages to read from the table on disk when the database server performs a read-ahead.

The RA_THRESHOLD parameter specifies the number of unprocessed data pages in memory that cause the database server to do another read-ahead. For example, if RA_PAGES is 10, and RA_THRESHOLD is 4, the database server reads ahead 10 data pages when 4 pages remain to be processed in the buffer.

For an example of the output that the **onstat -p** command produces to enable you to monitor the database server use of read-ahead, refer to "Monitoring the Shared-Memory Profile" on page 14-14 and to the utilities chapter in the *Administrator's Reference*.

The IDX_RA_PAGES parameter specifies the number of index pages to read from disk when the database server does a read-ahead of the index. The IDX_RA_THRESHOLD parameter specifies the number of unprocessed index pages in memory that cause the database server to do another read-ahead.

Database Server Thread Access to Buffer Pages

The database server uses shared-lock buffering to allow more than one database server thread to access the same buffer concurrently in shared memory. The database server uses two categories of buffer locks to provide this concurrency without a loss in thread isolation. The two categories of lock access are share and exclusive. (For more information, refer to "Types of Buffer Locks" on page 13-35.)

The process of accessing a data buffer consists of the following steps:

1. Identify the data requested by physical page number.
2. Determine the level of lock access needed by the thread for the requested buffer.
3. Attempt to locate the page in shared memory.
4. If the page is not in shared memory, locate a buffer in an FLRU queue, and read the page in from disk. If the page is in shared memory, proceed with step 5.
5. Proceed with processing, locking the buffer if necessary.
6. When finished accessing the buffer, release the lock.
7. Wake waiting threads with compatible lock-access types, if any exist.

Identify the Page

The database server threads request a specific data row, and the database server searches for the page that contains the row.

Determine the Level of Lock Access

Next the database server determines the requested level of lock access: share or exclusive.

Try to Locate the Page in Shared Memory

The thread first attempts to locate the requested page in shared memory. To do this, it acquires a mutex on the hash table that is associated with the buffer table. Then it searches the hash table to see if an entry matches the requested page. If the thread finds an entry for the page, it releases the mutex on the hash table and tries to acquire the mutex on the buffer entry in the buffer table.

The thread tests the current lock-access level of the buffer. If the levels are compatible, the requesting thread gains access to the buffer and sets its own lock. If the current lock-access level is incompatible, the requesting thread puts itself in the wait queue for the buffer.

The buffer state, unmodified or modified, is irrelevant to locking; even unmodified buffers can be locked.

If you configure the database server to use read-ahead, the database server performs a read-ahead request when the number of pages specified by the RA_THRESHOLD parameter remain to be processed in memory.

Locate a Buffer and Read Page from Disk

If the requested page must be read from disk, the thread first locates a usable buffer in the FLRU queues. The database server selects an FLRU queue at random and tries to acquire the mutex associated with the queue. If the mutex can be acquired, the buffer at the least-recently used end of the queue is used. If another thread holds the mutex, the first thread tries to acquire the mutex of another FLRU queue.

If you configure the database server to use read-ahead, the database server reads the number of pages specified by the RA_PAGES configuration parameter.

Lock the Buffer If Necessary

After a usable buffer is found, the buffer is temporarily removed from the FLRU queue. The thread creates an entry in the shared-memory buffer table as the page is read from disk into the buffer.

Release the Buffer Lock and Wake a Waiting Thread

When the thread is finished with the buffer, it releases the buffer lock. If any threads are waiting for the buffer, it wakes one up. However, this procedure varies, depending on whether the releasing thread modified the buffer.

When the Buffer Is Not Modified

If a thread does not modify the data, it releases the buffer as unmodified.

The release of the buffer occurs in steps. First, the releasing thread acquires the mutex on the buffer table that enables it to modify the buffer entry.

Next, it checks if other threads are sleeping, waiting for this buffer. If so, the releasing thread wakes the first thread in the wait queue that has a compatible lock-access type. The waiting threads are queued according to priorities that encompass more than just *first-come, first-served* hierarchies. (Otherwise, for example, threads waiting for exclusive access could wait forever.)

If no thread in the wait queue has a compatible lock-access type, any thread waiting for that buffer can receive access.

If no thread is waiting for the buffer, the releasing thread tries to release the buffer to the FLRU queue where it was found. If the latch for that FLRU queue is unavailable, the thread tries to acquire a latch for a randomly selected FLRU queue. When the FLRU queue latch is acquired, the unmodified buffer is linked to the most-recently used end of the queue.

After the buffer is returned to the FLRU queue, or the next thread in the wait queue is awakened, the releasing thread removes itself from the user list for the buffer and decrements the shared-user count by one.

When the Buffer Is Modified

If the thread intends to modify the buffer, to update a row in a table, for example, it acquires the mutex for the buffer and changes the buffer lock-access type to exclusive.

In most cases, a copy of the before-image of the page is needed for data consistency. If necessary, the thread determines whether a before-image of this page was written to either the physical-log buffer or the physical log since the last checkpoint. If not, a copy of the page is written to the physical-log buffer. Then the data in the page buffer is modified. If any transaction records are required for logging, those records are written to the logical-log buffer.

After the mutex for the buffer is released, the thread is ready to release the buffer. First, the releasing thread acquires the mutex on the buffer table that enables it to modify the buffer entry. Next, the releasing thread updates the time stamp in the buffer header so that the time stamp on the buffer page and the time stamp in the header match. Statistics describing the number and types of writes performed by this thread are updated.

The lock is released as described in the previous section, but the buffer is appended to the MLRU queue associated with the original FLRU queue.

Flushing Data to Disk

Writing a buffer to disk is called *buffer flushing*. When a user thread modifies data in a buffer, it marks the buffer as *dirty*. When the database server flushes the buffer to disk, it subsequently marks the buffer as *not dirty* and allows the data in the buffer to be overwritten.

The database server flushes the following buffers:

- Buffer pool (covered in this section)
- Physical-log buffer

  See "Flushing the Physical-Log Buffer" on page 13-46.
- Logical-log buffer

  See "Flushing the Logical-Log Buffer" on page 13-52.

Page-cleaner threads manage buffer flushing. The database server always runs at least one page-cleaner thread. If the database server is configured for more than one page-cleaner thread, the LRU queues are divided among the page cleaners for more efficient flushing. For information on specifying how many page-cleaner threads the database server runs, refer to the CLEANERS configuration parameter in the *Administrator's Reference*.

Flushing the physical-log buffer, the modified shared-memory page buffers, and the logical-log buffer must be synchronized with page-cleaner activity according to specific rules designed to maintain data consistency.

Events That Prompt Flushing of Buffer-Pool Buffers

Flushing of the buffers is initiated by any one of the following three conditions:

- The number of buffers in an MLRU queue reaches the number specified by LRU_MAX_DIRTY. (See "Flushing the Shared-Memory Pool Buffer" on page 13-50.)
- The page-cleaner threads cannot keep up. In other words, a user thread needs to acquire a buffer, but no unmodified buffers are available.
- The database server needs to execute a checkpoint. (See "Checkpoints" on page 24-4.)

Flushing Before-Images First

The overriding rule of buffer flushing is that the before-images of modified pages are flushed to disk before the modified pages themselves.

In practice, the physical-log buffer is flushed first and then the buffers that contain modified pages. Therefore, even when a shared-memory buffer page needs to be flushed because a user thread is trying to acquire a buffer, but none is available (a foreground write), the buffer pages cannot be flushed until the before-image of the page has been written to disk.

Flushing the Physical-Log Buffer

The database server temporarily stores before-images of some of the modified disk pages in the physical-log buffer. If the before-image has been written to the physical-log buffer but not to the physical log on disk, the physical-log buffer must be flushed to disk before the modified page can be flushed to disk. This action is required for the fast-recovery feature. Writing the before-image to the physical log buffer and then flushing the buffer page to disk is illustrated in Figure 13-9.

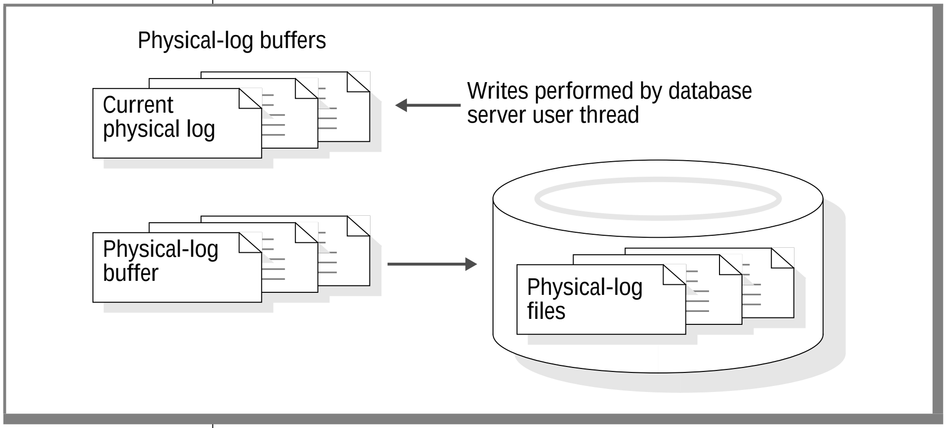


Figure 13-9
Physical-Log Buffer and Its Relation to the Physical Log on Disk

Both the physical-log buffer and the physical log contribute toward maintaining the physical and logical consistency of the data. For a description of physical logging, refer to Chapter 22, "Physical Logging." For a description of checkpoints and fast recovery, refer to Chapter 24, "Checkpoints and Fast Recovery."

Events That Prompt Flushing of the Physical-Log Buffer

The following four events cause the current physical-log buffer to flush:

- The current physical-log buffer becomes full.
- A modified page in shared memory must be flushed, but the before-image is still in the current physical-log buffer.
- A full or fuzzy checkpoint occurs.

The contents of the physical-log buffer must always be flushed to disk before any data buffers. This rule is required for the fast-recovery feature.

The database server uses only one of the two physical-log buffers at a time. This buffer is the current physical-log buffer. Before the database server flushes the current physical-log buffer to disk, it makes the other buffer the current buffer so that it can continue writing while the first buffer is being flushed.

When the Physical-Log Buffer Becomes Full

Buffer flushing that results from the physical-log buffer becoming full proceeds as follows.

When a user thread needs to write a before-image to the physical-log buffer, it acquires the mutex associated with the physical-log buffer and the mutex associated with the physical log on disk. If another thread is writing to the buffer, the incoming thread must wait for the mutexes to be released.

After the incoming thread acquires the mutexes, but before the write, the thread checks to see what percent of the physical log is full.

If the Log Is More Than 75 Percent Full

If the physical log is more than 75 percent full, the thread sets a flag to request a fuzzy checkpoint. Next, the thread claims the amount of space in the buffer that it needs for its write and releases the buffer mutex so that other threads can access the buffer. Finally, it copies the data to the space that it claimed in the buffer. The checkpoint does not begin until all user threads, including this one, are out of critical sections. For more information on critical sections and checkpoints, refer to "How the Database Server Achieves Data Consistency" on page 24-3.

If the Log Is Less Than 75 Percent Full

If the physical log is less than 75 percent full, the thread compares the page counter in the physical-log buffer header to the buffer capacity. If this one-page write does not fill the physical-log buffer, the thread reserves space in the log buffer for the write and releases the mutex. Any thread waiting to write to the buffer is awakened. After the thread releases the mutex, it writes the page to the reserved space in the physical-log buffer. The sequence of this operation increases concurrency and eliminates the need to hold the mutex during the write.

If this one-page write fills the physical-log buffer, flushing is initiated. First the page is written to the current physical-log buffer, filling it. Next, the thread latches the other physical-log buffer. The thread switches the shared-memory current-buffer pointer, making the newly latched buffer the current buffer. The mutex on the physical log on disk and the mutex on this new, current buffer are released, which permits other user threads to begin writing to the new current buffer. Last, the full buffer is flushed to disk, and the mutex on the buffer is released.

Each write to the physical-log buffer writes one page.

Synchronizing Buffer Flushing

When shared memory is first initialized, all buffers are empty. As processing occurs, data pages are read from disk into the buffers, and user threads begin to modify these pages.

Ensuring That Physical-Log Buffers Are Flushed First

When page cleaning is initiated on the shared-memory buffer pool, the page-cleaner thread must coordinate the flushing so that the physical-log buffer is flushed first. Time-stamp comparison determines the order.

The database server stores a time stamp each time that the physical-log buffer is flushed. If a page-cleaner thread needs to flush a page in a shared-memory buffer, the page cleaner compares the time stamp in the modified buffer with the time stamp that indicates the point when the physical-log buffer was last flushed.

If the time stamp on the page in the buffer pool is equal to or more recent than the time stamp for the physical-log buffer flush, the before-image of this page conceivably could be contained in the physical-log buffer. In this case, the physical-log buffer must be flushed before the shared-memory buffer pages are flushed.

Flushing the Shared-Memory Pool Buffer

After the physical-log buffer is flushed, the user thread updates the time stamp in shared memory that describes the most-recent physical-log buffer flush. The specific page in the shared-memory buffer pool that is marked for flushing is now flushed. The number of modified buffers in the queue is compared to the value of LRU_MIN_DIRTY. If the number of modified buffers is greater than the value represented by LRU_MIN_DIRTY, another page buffer is marked for flushing. The time-stamp comparison is repeated. If required, the physical-log buffer is flushed again.

When no more buffer flushing is required, the page-cleaner threads sleep until buffer flushing is required again, and they are awakened to do the work. (For more information, refer to "Sleep Queues" on page 11-15.) You can tune the page-cleaning parameters (LRU_MIN_DIRTY and LRU_MAX_DIRTY) to influence the frequency of buffer flushing. For a description of how these parameters determine when page cleaning begins and ends, refer to "LRU Queues" on page 13-36.

Describing Flushing Activity

To provide you with information about the specific condition that prompted buffer-flushing activity, the database server defines three types of writes and counts how often each write occurs:

- Foreground write
- LRU write
- Chunk write

To display the write counts that the database server maintains, use **onstat -F** as described in the utilities chapter of the *Administrator's Reference*.

If you implement mirroring for the database server, data is always written to the primary chunk first. The write is then repeated on the mirrored chunk. Writes to a mirrored chunk are included in the counts. For more information on monitoring the types of writes that the database server performs, refer to "Monitoring Buffer-Pool Activity" on page 14-18.

Foreground Write

Whenever an **sqlexec** thread writes a buffer to disk, it is termed a *foreground* write. A foreground write occurs when an **sqlexec** thread searches through the LRU queues on behalf of a user but cannot locate an empty or unmodified buffer. To make space, the **sqlexec** thread flushes pages, one at a time, to hold the data to be read from disk. (For more information, refer to "LRU Queues" on page 13-36.)

If the **sqlexec** thread must perform buffer flushing just to acquire a shared-memory buffer, performance can suffer. Foreground writes should be avoided. To display a count of the number of foreground writes, run **onstat -F**. If you find that foreground writes are occurring on a regular basis, tune the value of the page-cleaning parameters. Either increase the number of page cleaners or decrease the value of LRU_MAX_DIRTY.

LRU Write

Unlike foreground writes, LRU writes are performed by page cleaners rather than by **sqlexec** threads. The database server performs LRU writes as background writes that typically occur when the percentage of dirty buffers exceeds the percent you that specified in the LRU_MAX_DIRTY configuration parameter.

In addition, a foreground write can trigger an LRU write. When a foreground write occurs, the **sqlexec** thread that performed the write alerts a page-cleaner to wake up and clean the LRU for which it performed the foreground write.

In a properly tuned system, page cleaners ensure that enough unmodified buffer pages are available for storing pages to be read from disk. Thus, **sqlexec** threads that perform a query do not need to flush a page to disk before they read in the disk pages required by the query. This condition can result in significant performance gains for queries that do not make use of foreground writes.

LRU writes are preferred over foreground writes because page-cleaner threads perform buffer writes much more efficiently than **sqlexec** threads do. To monitor both types of writes, use **onstat -F**.

Chunk Write

Chunk writes are commonly performed by page-cleaner threads during a checkpoint or, possibly, when every page in the shared-memory buffer pool is modified. Chunk writes, which are performed as sorted writes, are the most efficient writes available to the database server.

During a chunk write, each page-cleaner thread is assigned to one or more chunks. Each page-cleaner thread reads through the buffer headers and creates an array of pointers to pages that are associated with its specific chunk. (The page cleaners have access to this information because the chunk number is contained within the physical page number address, which is part of the page header.) This sorting minimizes head movement (disk seek time) on the disk and enables the page-cleaner threads to use the big buffers during the write, if possible.

In addition, because user threads must wait for the checkpoint to complete, the page-cleaner threads are not competing with a large number of threads for CPU time. As a result, the page-cleaner threads can finish their work with less context switching.

Flushing the Logical-Log Buffer

The database server uses the shared-memory logical-log buffer as temporary storage for records that describe modifications to database server pages. From the logical-log buffer, these records of changes are written to the current logical-log file on disk and eventually to the logical-log backup media. For a description of logical logging, refer to Chapter 20, "Logical Log."

Five events cause the current logical-log buffer to flush:

- The current logical-log buffer becomes full.
- A transaction is prepared or committed in a database with unbuffered logging.
- A nonlogging database session terminates.
- A checkpoint occurs.
- A page is modified that does not require a before-image in the physical log.

The following sections discuss each of these events in detail.

When the Logical-Log Buffer Becomes Full

When a user thread needs to write records to the logical-log buffer, it acquires the mutexes associated with the logical-log buffer and the current logical log on disk. If another thread is writing to the buffer, the incoming thread must wait for the mutexes to be released.

After the incoming thread acquires the mutexes, but before the write, the thread checks how much logical-log space is available on disk. When the logical-log space on disk is full, and the database server switches to a new logical log, it checks if the percent of used log space is greater than the long-transaction high-water mark, specified by the LTXHWM parameter in the ONCONFIG file. For a description of this configuration parameter and information on specifying a value for it, refer to the *Administrator's Reference*.

If no long-transaction condition exists, the logical-log I/O thread compares the available space in the logical-log buffer with the size of the record to be written. If the write does not fill the logical-log buffer, the thread writes the record, releases latches, and awakens any threads that are waiting to write to the buffer.

If the write fills the logical-log buffer, flushing is initiated as follows:

1. The thread latches the next logical-log buffer. The thread then switches the shared-memory current-buffer pointer, making the newly latched buffer the current buffer.
2. The thread writes the new record to the new current buffer. The thread releases the latch on the logical log on disk and the latch on this current buffer, permitting other logical-log I/O threads to begin writing to this buffer.
3. The full logical-log buffer is flushed to disk, and the latch on the buffer is released. This logical-log buffer is now available for reuse.

After a Transaction Is Prepared or Terminated in a Database with Unbuffered Logging

If a transaction is prepared or terminated in a database with unbuffered logging, the logical-log buffer is immediately flushed. Flushing might cause a waste of some disk space. Typically, many logical-log records are stored on a single page. However, because the logical-log buffer is flushed in whole pages, the whole page is flushed even if only one transaction record is stored on the page. In the worst case, a single COMMIT logical-log record (COMMIT WORK) could occupy a page on disk, and all remaining space on the page would be unused. However, the cost in disk space of using unbuffered logging is minor compared to the benefits of insured data consistency.

The following log records cause flushing of the logical-log buffers in a database with unbuffered logging:

- COMMIT
- PREPARE
- XPREPARE
- ENDTRANS

For a comparison of buffered versus unbuffered logging, refer to the SET LOG statement in the *Informix Guide to SQL: Syntax*.

When a Session That Uses Nonlogging Databases or Unbuffered Logging Terminates

Even for nonlogging databases, the database server logs certain activities that alter the database schema, such as the creation of tables or extents. When the database server terminates sessions that use unbuffered logging or nonlogging databases, the logical-log buffer is flushed to make sure that any logging activity is recorded.

When a Checkpoint Occurs

For a detailed description of the events that occur during a checkpoint, refer to "Checkpoints" on page 24-4.

When a Page Is Modified That Does Not Require a Before-Image in the Physical-Log File

When a page is modified that does not require a before-image in the physical log, the logical-log buffer must be flushed before that page is flushed to disk.

Blobpages are allocated and tracked with the free-map page. Links that connect the blobpages and pointers to the next blobpage segments are created as needed.

A record of the operation (insert, update, or delete) is written to the logical-log buffer.

Buffering Simple-Large-Object Data Types

TEXT and BYTE data types pass through the buffer pool if the table is a logging table.

For SELECT, INSERT, UPDATE, and DELETE operations, the database server writes TEXT and BYTE data to disk pages in a dbspace in the same way that it writes any other data type. For more information, refer to "Flushing Data to Disk" on page 13-45.

When you load with external tables, the TEXT and BYTE data can be in two different formats:

- Informix internal format

  For Informix internal format, the database server places simple large objects after the row rather than in it.
- Delimited format

  For delimited format, the simple large objects are embedded within the row. When loaded from a pipe, the data might be staged to a temporary area on disk because it does not fit into the buffer pool.

For more information, refer to the loading with external tables chapter in the *Administrator's Reference*.

Blobpages Do Not Pass Through Shared Memory

Blobpages store large amounts of data. Consequently, the database server does not create or access blobpages by way of the shared-memory buffer pool, and it does not write blobpages to either the logical or physical logs.

If blobpage data passed through the shared-memory pool, it has the potential to dilute the effectiveness of the pool by driving out index pages and data pages. Instead, blobpage data is written directly to disk when it is created.

To reduce logical-log and physical-log traffic, the database server writes blobpages from magnetic media to dbspace backup tapes and logical-log backup tapes in a different way than it writes dbspace pages. For a description of how dbspaces are logged, refer to "Dbspace Logging" on page 20-21.

TEXT and BYTE Objects Are Created Before the Data Row Is Inserted

When TEXT or BYTE data is written to disk, the row to which it belongs might not exist yet. During an insert, for example, the TEXT or BYTE data is transferred before the rest of the row data. After the TEXT or BYTE object is stored, the data row is created with a 56-byte descriptor that points to its location. For a description of how TEXT and BYTE data types are stored physically, refer to the section on the structure of a dbspace blobpage in the disk storage and structure chapter of the *Administrator's Reference*.

Tracking Blobpages

Blobpages are allocated and tracked using the free-map page. Links that connect the blobpages and pointers to the next blobpage segments are created as needed.

A record of the operation (insert, update, or delete) is written to the logical-log buffer.

Memory Use on 64-Bit Platforms

Because 64-bit platforms allow for larger memory-address space, the maximum values for the following memory-related configuration parameters are larger on 64-bit platforms:

- BUFFERS
- CLEANERS
- DS_MAX_QUERIES
- DS_TOTAL_MEMORY
- LOCKS
- LRUS
- SHMADD
- SHMVIRTSIZE

The following configuration parameters allow noninteger values on 64-bit platforms:

- LRU_MAX_DIRTY
- LRU_MIN_DIRTY

For more information about the minimum and maximum values for these parameters, consult your *Administrator's Reference*.

Chapter

Managing Shared Memory

14

In This Chapter

This chapter tells you how to perform tasks related to managing the use of shared memory with the database server. It assumes you are familiar with the terms and concepts in Chapter 13, "Shared Memory."

This chapter describes how to perform the following tasks:

- Set the shared-memory configuration parameters
- Reinitialize shared memory
- Turn residency on or off for the resident portion of the database server shared memory
- Add a segment to the virtual portion of shared memory
- Monitor shared memory

This chapter does not cover the DS_TOTAL_MEMORY configuration parameter. This parameter places a ceiling on the allocation of memory for decision-support queries. For information on this parameter, refer to your *Performance Guide*.

Each coserver has its own shared memory. The shared memory as discussed in this chapter is for an individual coserver.

Setting Operating-System Shared-Memory Configuration Parameters

Several operating-system configuration parameters can affect the use of shared memory by the database server.Parameter names are not provided because names vary among platforms, and not all parameters exist on all platforms. The following list describes these parameters by function:

- Maximum operating-system shared-memory segment size, expressed in bytes or kilobytes
- Minimum shared-memory segment size, expressed in bytes
- Maximum number of shared-memory identifiers
- Lower-boundary address for shared memory
- Maximum number of attached shared-memory segments per process
- Maximum amount of systemwide shared memory
- Maximum number of semaphore identifiers
- Maximum number of semaphores
- Maximum number of semaphores per identifier

On UNIX, the machine notes file contains recommended values that you use to configure operating-system resources. Use these recommended values when you configure the operating system. For information on how to set these operating-system parameters, consult your operating-system manuals.

For specific information about your operating-system environment, refer to the machine notes file that is provided with the database server. For more information about the machine notes file, refer to "Documentation Notes, Release Notes, Machine Notes" on page 12 in the Introduction. ♦

Maximum Shared-Memory Segment Size

When the database server creates the required shared-memory segments, it attempts to acquire as large an operating-system segment as possible. The first segment size that the database server tries to acquire is the size of the portion that it is allocating (resident, virtual, or communications), rounded up to the nearest multiple of 8 kilobytes.

The database server receives an error from the operating system if the requested segment size exceeds the maximum size allowed. If the database server receives an error, it divides the requested size by two and tries again. Attempts at acquisition continue until the largest segment size that is a multiple of 8 kilobytes can be created. Then the database server creates as many additional segments as it requires.

Maximum Number of Shared-Memory Identifiers

Shared-memory identifiers affect the database server operation when a virtual processor attempts to attach to shared memory. The operating system identifies each shared-memory segment with a shared-memory identifier. For most operating systems, virtual processors receive identifiers on a *first-come, first-served* basis, up to the limit that is defined for the operating system as a whole. For more information about shared-memory identifiers, refer to "How Virtual Processors Attach to Shared Memory" on page 13-13.

You might be able to calculate the maximum amount of shared memory that the operating system can allocate by multiplying the number of shared-memory identifiers by the maximum shared-memory segment size.

Shared-Memory Lower-Boundary Address

When the database server attaches shared-memory segments subsequent to the first segment, it assumes that the segment can be attached contiguous with the previous one; that is, that a segment can be attached at the address of the previous segment plus the size of that segment. However, your operating system might set a parameter that defines a lower-boundary address for attaching shared-memory segments. If the size of a segment would cause it to cross the lower-boundary address, the segment is attached at a point beyond the end of the previous segment, creating a gap between shared-memory segments. For an illustration of this situation, refer to "How Virtual Processors Attach to Shared Memory" on page 13-13.

Maximum Amount of Shared Memory for One Process

Check that the maximum amount of memory that can be allocated for one process is equal to the total addressable shared-memory size for a single operating-system process. The following equation expresses the concept another way:

```
Maximum amount of shared memory for one process =
    (Maximum number of attached shared-memory segments per
    process) x (Maximum shared-memory segment size)
```

If this relationship does not hold, one of two undesirable situations could develop:

- If the total amount of shared memory is less than the total addressable shared-memory size, you can address more shared memory for the operating system than is available.
- If the total amount of shared memory is greater than the total addressable size of shared memory, you can never address some amount of shared memory that is available. That is, space that could potentially be used as shared memory cannot be allocated.

Setting Database Server Shared-Memory Configuration Parameters

Shared-memory configuration parameters fall into the following categories based on their purposes:

- Parameters that affect the resident portion of shared memory
- Parameters that affect the virtual portion of shared memory
- Parameters that affect performance

You can set shared-memory configuration parameters using a text editor. You must be **root** or user **informix**.

Before any changes that you make to the configuration parameters take effect, you must reinitialize shared memory by stopping and starting the database server.

Setting Parameters for Resident Shared Memory with a Text Editor

You can use a text editor to set shared-memory configuration parameters at any time. Use the editor to locate the parameter in the ONCONFIG file, enter the new value or values, and rewrite the file to disk. Before the changes take effect, however, you must reinitialize shared memory.

Figure 14-1 lists the parameters in the ONCONFIG file that specify the configuration of the buffer pool and the internal tables in the resident portion of shared memory. For a description of the configuration parameters, refer to the *Administrator's Reference*.

Figure 14-1
Configuring the Resident Portion of Shared Memory

ONCONFIG Parameter	Purpose
BUFFERS	Specifies the maximum number of shared-memory buffers
ISOLATION_LOCKS	Specifies the maximum number of rows that can be locked on a single scan when Cursor Stability isolation level is in effect. For performance considerations when using this ISOLATION_LOCKS parameter, refer to your *Performance Guide*.
LOCKS	Specifies the initial number of locks for database objects; for example, rows, key values, pages, and tables
LOGBUFF	Specifies the size of the logical-log buffers
PHYSBUFF	Specifies the size of the physical-log buffers
RESIDENT	Specifies residency for the resident portion of the database server shared memory
SERVERNUM	Specifies a unique identification number for the database server on the local host computer
SHMTOTAL	Specifies the total amount of memory to be used by the database server

Setting Parameters for Virtual Shared Memory with a Text Editor

You can use a text editor at any time to set the virtual shared-memory configuration parameters. Use the editor to locate the parameter in the file, enter the new value or values, and rewrite the file to disk.

Figure 14-2 lists the ONCONFIG parameters that you use to configure the virtual portion of shared memory. For more information, see the chapter on configuration effects on memory in your *Performance Guide*.

Figure 14-2
Configuring the Virtual Portion of Shared Memory

ONCONFIG Parameter	Purpose
PC_POOLSIZE	Specifies the number of SPL routines that can be stored in the SPL routine cache.
PC_HASHSIZE	Specifies the number of hash buckets in the SPL routine cache.
SHMVIRTSIZE	Specifies the initial size of the virtual portion of shared memory
STACKSIZE	Specifies the stack size for the database server user threads
SHMADD	Specifies the size of dynamically added shared-memory segments
SHMTOTAL	Specifies the total amount of memory to be used by the database server

Setting Parameters for Shared-Memory Performance Options with a Text Editor

You can use a text editor to set ONCONFIG parameters at any time. To change one of the configuration parameters that set shared-memory performance options, use the text editor to locate the parameter in the file, enter the new value or values, and rewrite the file to disk. The changes that you make do not take effect until you reinitialize shared memory.

Figure 14-3 on page 14-10 lists the ONCONFIG parameters that set shared-memory performance options. For more information, see the chapter on configuration parameters in the *Administrator's Reference*.

Figure 14-3
Setting Shared-Memory Performance Options

ONCONFIG Parameter	Purpose
CKPTINTVL	Specifies the maximum number of seconds that can elapse before the database server checks if a checkpoint is needed
CLEANERS	Specifies the number of page-cleaner threads that the database server is to run
LRU_MAX_DIRTY	Specifies the percentage of modified pages in the LRU queues that flags page cleaning to start
LRU_MIN_DIRTY	Specifies the percentage of modified pages in the LRU queues that flags page cleaning to stop
LRUS	Specifies the number of LRU queues for the shared-memory buffer pool
RA_PAGES	Specifies the number of disk pages that the database server should attempt to read ahead when it performs sequential scans of data or index records
RA_THRESHOLD	Specifies the number of unprocessed memory pages that, after they are read, cause the database server to read ahead on disk
IDX_RA_PAGES	Specifies the number of disk pages that the database server should attempt to read ahead when it performs sequential scans of index records
IDX_RA_THRESHOLD	Specifies the number of unprocessed memory pages that, after they are read, cause the database server to read ahead more index pages on disk

Reinitializing Shared Memory

The database server reinitializes shared memory when you take the database server from off-line mode to quiescent mode or when you take it from off-line mode directly to on-line mode. To reinitialize shared memory, first bring the database server off-line. After the database server is off-line, bring it to quiescent mode or on-line mode to reinitialize shared memory. For information on how to take the database server from on-line mode to off-line, refer to Chapter 9, "Managing Database Server Operating Modes."

Turning Residency On or Off for Resident Shared Memory

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

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

For a description of the resident portion of shared memory, refer to "Resident Portion of Shared Memory" on page 13-19.

Turning Residency On or Off in On-Line Mode

To turn residency on or off while the database server is in on-line mode, use the **onmode** utility.

To turn on residency immediately for the resident portion of shared memory, execute the following command:

```
% onmode -r
```

To turn off residency immediately for the resident portion of shared memory, execute the following command:

```
% onmode -n
```

These commands do not change the value of the RESIDENT parameter in the ONCONFIG file. That is, this change is not permanent, and residency reverts to the state specified by the RESIDENT parameter the next time that you initialize shared memory. You cannot use the **onmode -r** command to turn residency on or off unless the RESIDENT parameter is set in your ONCONFIG file when you initialize the database server memory.

On UNIX, you must be **root** or user **informix** to turn residency on or off. On Windows NT, you must be a user in the **Informix Admin** group to turn residency on or off.

Turning Residency On or Off for the Next Time You Reinitialize Shared Memory

You can use a text editor to turn residency on or off for the next time that you reinitialize shared memory. To change the current state of residency, use a text editor to locate the RESIDENT parameter. Set RESIDENT to `1` to turn residency on or to `0` to turn residency off, and rewrite the file to disk. Before the changes take effect, you must reinitialize shared memory.

Adding a Segment to the Virtual Portion of Shared Memory

The **-a** option of the **onmode** utility allows you to add a segment of specified size to virtual shared memory.

You do not normally need to add segments to virtual shared memory because the database server automatically adds segments as needed.

The option to add a segment with the **onmode** utility is useful if the number of operating-system segments is limited, and the initial segment size is so low, relative to the amount that is required, that the operating-system limit of shared-memory segments is nearly exceeded.

Monitoring Shared Memory

This section describes how to monitor shared-memory segments, the shared-memory profile, and the use of specific shared-memory resources (buffers, latches, and locks).

You can use the **onstat -o** utility to capture a static snapshot of database server shared memory for later analysis and comparison.

Monitoring Shared-Memory Segments

Monitor the shared-memory segments to determine the number and size of the segments that the database server creates. The database server allocates shared-memory segments dynamically, so these numbers can change. If the database server is allocating too many shared-memory segments, you can increase the SHMVIRTSIZE configuration parameter. For more information, see the chapter on configuration parameters in the *Administrator's Reference*.

The **onstat -g seg** command lists information for each shared-memory segment, including the address and size of the segment. Figure 14-4 shows sample output.

Figure 14-4
onstat -g seg Output

```
Segment Summary:
 (resident segments are not locked)
id       key        addr     size       ovhd      class blkused   blkfree
300      1381386241 400000   614400     800       R     71        4
301      1381386242 496000   4096000    644       V     322       178
```

Monitoring the Shared-Memory Profile

Monitor the database server profile to analyze performance and the use of shared-memory resources. The Profile screen maintains cumulative statistics on shared-memory use. To reset these statistics to zero, use the **onstat -z** option.

Using Command-Line Utilities

Execute **onstat -p** to display statistics on database server activity. Figure 14-5 shows these statistics.

Figure 14-5
onstat -p Output

```
Profile
dskreads pagreads bufreads %cached dskwrits pagwrits bufwrits %cached
382      400      14438    97.35   381      568      3509     89.14

isamtot  open     start    read     write    rewrite  delete   commit   rollbk
9463     1078     1584     2316     909      162      27       183      1

ovlock   ovuserthread ovbuff   usercpu  syscpu   numckpts flushes
0        0            0        13.55    13.02    5        18

bufwaits lokwaits lockreqs deadlks  dltouts  ckpwaits compress seqscans
14       0        16143    0        0        0        101      68

ixda-RA  idx-RA   da-RA    RA-pgsused lchwaits
5        0        204      148        12
```

The **onstat -p** output contains several fields that are not included in the information that the ON-Monitor Profile option displays. For a description of all the fields that **onstat** displays, see the utilities chapter in the *Administrator's Reference*.

Using SMI Tables

Query the **sysprofile** table to obtain shared-memory statistics. This table contains all of the statistics available in **onstat -p** output except the **ovbuff**, **usercpu**, and **syscpu** statistics.

Monitoring Buffers

You can obtain both statistics on buffer use and information on specific buffers.

The statistical information includes the percentage of data writes that are cached to buffers and the number of times that threads had to wait to obtain a buffer. The percentage of writes cached is an important measure of performance. (For information on how to use this statistic to tune the database server, see your *Performance Guide*.) The number of waits for buffers gives a measure of system concurrency.

Information on specific buffers includes a listing of all the buffers in shared memory that are held by a thread. This information allows you to track the status of a particular buffer. For example, you can determine if another thread is waiting for the buffer.

Using Command-Line Utilities

You can use the following command-line utilities to monitor buffers:

- **onstat -p**
- **onstat -B**
- **onstat -b**
- **onstat -X**

onstat -p

Execute **onstat -p** to obtain statistics about cached reads and writes. The following caching statistics appear in four fields on the top row of the output display:

- The number of reads from shared-memory buffers (**bufreads**)
- The percentage of reads cached (**%cached**)
- The number of writes to shared memory (**bufwrits**)
- The percentage of writes cached (**%cached**)

Figure 14-6 shows these fields.

```
Profile
dskreads pagreads bufreads %cached dskwrits pagwrits bufwrits %cached
382      400      14438    97.35   381      568      3509     89.14
...
```

Figure 14-6
Cached Read and Write Statistics in the onstat -p Output

The number of reads or writes can appear as a negative number if the number of occurrences exceeds 2^{32}.

The **onstat -p** option also displays a statistic (**bufwaits**) that indicates the number of times that sessions had to wait for a buffer.

onstat -B

Execute **onstat -B** to obtain the following buffer information:

- Address of every regular shared-memory buffer
- Page numbers for all pages that remain in shared memory
- Address of the thread that currently holds the buffer
- Address of the first thread that is waiting for each buffer

Figure 14-7 shows an example of **onstat -B** output.

```
Buffers
address   userthread flgs pagenum   memaddr   nslots pgflgs xflgs owner     waitlist
849ae8    0          86   100955    84e000    1      b0     0     0         0
849b40    0          6    10095b    84e800    0      4      0     0         0
849b98    0          6    1009eb    84f000    0      4      0     0         0
849bf0    0          6    1008f5    84f800    2      70     0     0         0
...
84dea0    0          86   10093e    8b0800    8      1      0     0         0
84def8    0          6    10094b    8b1000    0      4      0     0         0
84df50    0          86   1009cd    8b1800    9      b0     0     0         0
 0 modified, 200 total, 256 hash buckets, 2048 buffer size
```

Figure 14-7
onstat -B Output

onstat -b

Execute **onstat -b** to obtain the following information about each buffer:

- Address of each buffer currently held by a thread
- Page numbers for the page held in the buffer
- Type of page held in the buffer (for example, data page, tblspace page, and so on)
- Type of lock placed on the buffer (exclusive or shared)
- Address of the thread that is currently holding the buffer
- Address of the first thread that is waiting for each buffer

You can compare the addresses of the user threads to the addresses that appear in the **onstat -u** display to obtain the session ID number. Figure 14-8 shows sample output. For more information on the fields that **onstat** displays, see the utilities chapter of the *Administrator's Reference*.

```
Buffers
address   userthread flgs pagenum   memaddr   nslots pgflgs xflgs owner     waitlist
84a748    0          27   1012b0    860000    19     2001   80    8067c4    0
84add0    0          0    101752    869800    19     2001   80    807890    0
84b2a0    0          27   100c31    870800    19     2001   80    8067c4    0
84c798    0          27   10108e    88f000    19     2001   80    8067c4    0
84d818    0          27   101272    8a7000    19     2001   80    8067c4    0
 154 modified, 200 total, 256 hash buckets, 2048 buffer size
```

Figure 14-8
onstat -b Output

onstat -X

Execute **onstat -X** to obtain the same information as for **onstat -b**, along with the *complete* list of all threads that are waiting for buffers, not just the first waiting thread.

Using SMI Tables

Query the **sysprofile** table to obtain statistics on cached reads and writes and total buffer waits. The following rows are relevant.

Row	Description
dskreads	Number of reads from disk
bufreads	Number of reads from buffers
dskwrites	Number of writes to disk
bufwrites	Number of writes to buffers
buffwts	Number of times that any thread had to wait for a buffer

Monitoring Buffer-Pool Activity

You can obtain statistics that relate to buffer availability as well as information on the buffers in each LRU queue.

The statistical information includes the number of times that the database server attempted to exceed the maximum number of buffers and the number of writes to disk (categorized by the event that caused the buffers to flush). These statistics help you determine if the number of buffers is appropriate. For information on tuning database server buffers, see your *Performance Guide*.

Information on the buffers in each LRU queue consists of the length of the queue and the percentage of the buffers in the queue that have been modified.

Using Command-Line Utilities

You can use the **onstat** command-line utility to obtain information on buffer-pool activity. For more information about the **onstat** options, refer to the utilities chapter of the *Administrator's Reference*.

onstat -p

The **onstat -p** output contains a statistic (**ovbuff**) that indicates the number of times the database server attempted to exceed the maximum number of shared buffers specified by the BUFFERS parameter in the ONCONFIG file. Figure 14-9 shows **onstat -p** output, including the **ovbuff** field.

```
...
ovtbls    ovlock    ovuserthread ovbuff    usercpu   syscpu    numckpts flushes
0         0         0            0         13.55     13.02     5        18
...
```

Figure 14-9
onstat -p Output Showing ovbuff Field

onstat -F

Execute **onstat -F** to obtain a count by write type of the writes performed. (For an explanation of the different write types, see "Describing Flushing Activity" on page 13-50.) Figure 14-10 on page 14-20 shows an example of the output. This information tells you when and how the buffers are flushed.

The **onstat -F** command displays totals for the following write types:

- Foreground write
- LRU write
- Chunk write

The **onstat -F** command also lists the following information about the page cleaners:

- Page-cleaner number
- Page-cleaner shared-memory address
- Current state of the page cleaner
- LRU queue to which the page cleaner was assigned

Figure 14-10 shows an example of the **onstat -F** output.

```
...
Fg Writes     LRU Writes    Chunk Writes
0             146           140

address  flusher  state    data
8067c4   0        I        0        = 0X0
      states: Exit Idle Chunk Lru
```

Figure 14-10
onstat -F Output

onstat -R

Execute **onstat -R** to obtain information about the number of buffers in each LRU queue and the number and percentage of the buffers that are modified or free. Figure 14-11 shows an example of **onstat -R** output.

```
8 buffer LRU queue pairs
# f/m   length   % of   pair total
 0 f        3    37.5%       8
 1 m        5    55.6%
 2 f        5    45.5%      11
 3 m        6    54.5%
 4 f        2    18.2%      11
 5 m        9    81.8%
 6 f        5    50.0%      10
 7 m        5    55.6%
 8 F        5    50.0%      10
 9 m        5    45.5%
10 f        0     0.0%      10
11 m       10   100.0%
12 f        1    11.1%       9
13 m        8    88.9%
14 f        2    28.6%       7
15 m        5    71.4%
53 dirty, 76 queued, 80 total, 128 hash buckets, 2048 buffer size
start clean at 60% (of pair total) dirty, or 6 buffs dirty, stop at 50%
```

Figure 14-11
onstat -R Output

Using SMI Tables

Query the **sysprofile** table to obtain the statistics on write types that are held in the following rows.

Row	Description
fgwrites	Number of foreground writes
lruwrites	Number of LRU writes
chunkwrites	Number of chunk writes

Monitoring Latches

You can obtain statistics on latch use and information on specific latches.

The statistics include the number of requests for latches and the number of times that threads had to wait to obtain a latch. These statistics provide a measure of the system activity.

Information on specific latches includes a listing of all the latches that are held by a thread and any threads that are waiting for latches. This information allows you to locate any specific resource contentions that exist.

Using Command-Line Utilities

You can use the following command-line utilities to obtain information about latches.

onstat -p

Execute **onstat -p** to obtain the values in the fields **lchreqs** and **lchwaits**. These fields store the number of requests for a latch and the number of times that a thread was required to wait for a shared-memory latch. A large number of latch waits typically results from a high volume of processing activity in which the database server is logging most of the transactions. (The administrator cannot configure or tune the number of latches; the database server sets this function internally.) Figure 14-12 shows **onstat -p** output, including the **lchreqs** and **lchwaits** fields.

```
...
ixda-RA  idx-RA   da-RA    RA-pgsused lchreqs  lchwaits
5        0        204      148        151762   12
```

Figure 14-12
onstat -p Output Showing lchwaits Field

onstat -s

Execute **onstat -s** to obtain general latch information. The output includes the **userthread** column, which lists the address of any user thread that is waiting for a latch. (See Figure 14-13.) You can compare this address with the user addresses in the **onstat -u** output to obtain the user-process identification number.

Warning: *Never kill a database server process that is holding a latch. If you do, the database server immediately initiates an abort.*

```
Latches with lock or userthread set
name     address  lock wait userthread
LRU1     402e90   0    0        6b29d8
bf[34]   4467c0   0    0        6b29d8
```

Figure 14-13
onstat -s Output

Using SMI Tables

Query the **sysprofile** table to obtain the number of requests for a latch and the number of times a thread had to wait for a latch. The following rows are relevant.

Row	Description
latchreqs	Number of requests for a latch
latchwts	Number of times that a thread had to wait for a latch

Chapter

Data Storage

15

In This Chapter

This chapter defines terms and explains the concepts that you must understand to perform the tasks described in Chapter 16, “Managing Disk Space.” This chapter covers the following topics:

- Definitions of the physical and logical units that the database server uses to store data on disk
- Instructions on how to calculate the amount of disk space that you need to store your data
- Guidelines on how to lay out your disk space and where to place your databases and tables

The release notes file contains supplementary information on the maximum values related to the storage units discussed in this chapter. For information on how to access this file, see “Documentation Notes, Release Notes, Machine Notes” on page 12 of the Introduction.

Overview of Data Storage

The database server uses the following physical units to manage disk space:

- Chunk
- Page
- Extent

Overlying the physical units of storage space, the database server supports the following logical units associated with database management:

- Dbspace
- Dbslice
- External table
- Database
- Table
- Tblspace

The database server maintains the following additional disk-space storage structures to ensure physical and logical consistency of data:

- Logical log
- Physical log
- Reserved pages

Because these additional disk-space structures are not permanent storage units, they are not described in this chapter. For information about the logical log, see Chapter 20, "Logical Log." For information about the physical log, see Chapter 22, "Physical Logging." For information about reserved pages, see the disk structures and storage chapter in the *Administrator's Reference*.

The following sections describe the various data-storage units that the database server supports and the relationships between those units.

Physical Units of Storage

The database server uses the physical units of storage to allocate disk space. Unlike the logical units of storage whose size fluctuates, each of the physical units has a fixed or assigned size that is determined by the disk architecture.

The following sections describe the physical units of storage in more detail.

Chunks

A *chunk* is the largest unit of physical disk dedicated to database server data storage. Chunks provide administrators with a conveniently large unit for allocating disk space.

Some operating systems use the concept of a logical volume, and others use a logical unit. Each of these terms represents the smallest unit of physical disk that you can assign. A database server chunk is the same as a logical volume or a logical unit.

Uses of Chunks

The database server administrator assigns one or more chunks to dbspaces, the logical storage spaces that the database server supports.

The database server administrator typically adds a chunk to these storage spaces when that storage space approaches full capacity. For more information on these logical storage spaces, refer to "Logical Units of Storage" on page 15-14.

The database server also uses chunks for mirroring. A *primary chunk* is a chunk from which the database server copies data to a *mirrored chunk*. If the primary chunk fails, the database server brings the mirrored chunk on-line automatically. For more information on mirroring, see Chapter 25, "Mirroring."

Chunk Size, Number, and Names

Chunk names follow the same rules as dbspace names. For more details, see "Naming Chunks and Storage Spaces" on page 16-21.

For information on the maximum size and number of chunks that you can allocate on the database server system, see "Limiting Chunk Size and Number" on page 16-22.

Disk Allocation for Chunks

The database server can use regular operating-system files to store data. On operating systems that support raw disks, the database server can also use raw disk space to store data. Informix recommends that you use raw disks to store data whenever performance or data consistency is important.

Unbuffered or Buffered Disk Access on UNIX

You can allocate disk space in two ways:

- Use files that are buffered through the operating system, also referred to as *cooked* files.
- Use unbuffered disk access.

Unbuffered disk access can be through a raw disk device, or character-special files. As a general guideline, you experience better performance and increased reliability when you use unbuffered file access.

On UNIX, the raw disk interface that character-special files provide yields significant performance advantages. I/O to raw disk bypasses the buffering operations that the operating system performs on regular (cooked) files.

Raw Disk Space on UNIX

UNIX uses the concept of a *device* to describe peripherals such as magnetic disks and tapes, terminals, and communication lines. One type of device is a *block device*, such as a hard disk or a tape. A block device can be configured with an interface that provides buffering or with a raw interface that leaves the buffering to the application. When you configure a block device with a raw interface, the device is called a *raw device*, and the storage space that the device provides is called *raw disk space*. Space in a chunk of raw disk space is physically contiguous.

A raw interface is also referred to as a *character-special device*. The name of the chunk is the name of the character-special file in the **/dev** directory. In many operating systems, you can distinguish the character-special file from the block-special file by the first letter in the filename (typically *r*). For example, **/dev/rsd0f** is the character-special device that corresponds to the **/dev/sd0f** block-special device.

Cooked Files

A cooked file is a regular file that the operating system manages. Although the database server manages the contents of cooked files, the operating system manages all I/O to cooked files. Unlike raw disk space, the logically contiguous blocks of a cooked file might not be physically contiguous.

Even though a cooked file is a regular file, the database server manages the internal arrangement of data within the file. Never edit the contents of a cooked file that the database server manages. To do so puts the integrity of your data at risk.

Data Management with Cooked Files Versus Raw Disk Devices

When the operating system reads from a cooked file, it reads the data from disk to an internal buffer pool. Later, a second copy operation copies it from the operating system to the location requested by the application. Therefore. when two users both read the same file, the data is read from disk only once but copied from the operating-system buffer twice.

By contrast, when the operating system reads data from an unbuffered file or a raw disk device, it bypasses the operating-system buffer pool and copies the data directly to the location requested by the application. The database server requests that the data be placed in shared memory, making it immediately available to all database server virtual processors and running threads with no further copying.

Unbuffered Disk Access

A raw device or unbuffered file can directly transfer data between shared memory and the disk with direct memory access (DMA), which results in better performance by orders of magnitude.

When you use a raw device or unbuffered file to store your data, the database server guarantees that committed data is stored on disk. (The next section explains why no such guarantee can be made when you use cooked files to store your data.)

When you decide to allocate raw disk space to store your data, you must take the following steps:

1. Create and install a raw device.
2. Change the ownership and permissions of the device.

For more information on these steps, see "Allocating Raw Disk Space on UNIX" on page 16-10.

Use of Cooked Files

You can more easily allocate cooked files than raw disk space. To allocate raw space, you must have a disk partition available that is dedicated to raw space. To allocate a cooked file, you need only create the file on any existing partition. However, you sacrifice reliability and might experience diminished performance when you store the database server data in cooked files.

The buffering mechanism that most operating systems provide can become a performance bottleneck. If you must use cooked UNIX files, store the least frequently accessed data in those files. Store the files in a file system located near the center cylinders of the disk device or in a file system with minimal activity.

In a learning environment, where reliability and performance are not critical, cooked files can be convenient.

When performance is not a consideration, you can also use cooked files for static data (which seldom or never changes). Such data is less vulnerable to the problems associated with UNIX buffering in the event of a system failure.

When a chunk consists of cooked disk space, the name of the chunk is the complete pathname of the file. Because the chunk of cooked disk space is an operating-system file, space in the chunk might not be physically contiguous.

Warning: *Cooked files are less reliable than raw disk space because the operating system manages I/O for a cooked file. A write to a cooked file can result in data being written to a memory buffer in the operating-system file manager instead of being written immediately to disk. As a consequence, the database server cannot guarantee that the committed data actually reaches the disk. Database server recovery depends on the guarantee that data written to disk is actually on disk. In the event of system failure, if the data is not present on disk, the database server automatic-recovery mechanism might not be able to execute properly. The end result would be inconsistent data.*

When you decide to allocate cooked space to store your data, you must take the following steps:

1. Create a cooked file.
2. Change the ownership and permissions.

These steps are described in detail in "Allocating a File for Disk Space on UNIX" on page 16-8.

Offsets

The system administrator might divide a physical disk into *partitions*, which are different parts of a disk that have separate pathnames. Although Informix recommends that you use an entire disk partition when you allocate a chunk on a raw disk device, you can subdivide partitions or cooked files into smaller chunks using *offsets*. For more information, see "Strive to Associate Partitions with Chunks" on page 15-40.

An offset allows you to indicate the number of kilobytes into a device or cooked file to reach a given chunk. For example, suppose that you create a 1000 kilobyte chunk that you want to divide into two chunks of 500 kilobytes each. You can use an offset of zero kilobytes to mark the beginning of the first chunk and an offset of 500 kilobytes to mark the beginning of the second chunk.

You can specify an offset whenever you create, add, or drop a chunk from a a dbspace.

On Extended Parallel Server, the maximum chunk size and offset can be 4 gigabytes or even larger for 64-bit platforms. To determine which chunk size your platform supports, refer to your machine notes file.

You might also need to specify an offset to prevent the database server from overwriting partition information. "Allocating Raw Disk Space on UNIX" on page 16-10 explains when and how to specify an offset.

Pages

A *page* is the physical unit of disk storage that the database server uses to read from and write to Informix databases. Figure 15-1 illustrates the concept of a page, represented by a darkened sector of a disk platter.

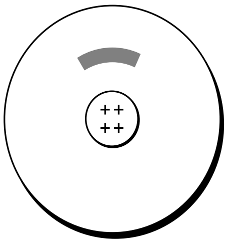


Figure 15-1
A Page on Disk

The default page size is 4 kilobytes. Use the PAGESIZE parameter to configure a page size of 2, 4, or 8 kilobytes. For more information on PAGESIZE, see "Configuring the Database Server Page Size" on page 3-17 and the chapter on configuration parameters in the *Administrator's Reference*.

A chunk contains a certain number of pages, as Figure 15-2 illustrates. A page is always entirely contained within a chunk; that is, a page cannot cross chunk boundaries.

For information on how the database server structures data within a page, see the chapter on disk structures and storage in the *Administrator's Reference*.

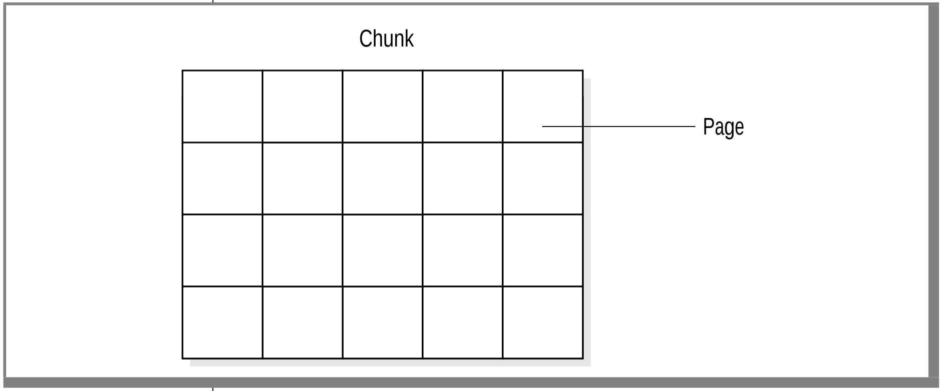


Figure 15-2
A Chunk, Logically Separated into a Series of Pages

Extents

When you create a table, the database server allocates a fixed amount of space to contain the data to be stored in that table. When this space fills, the database server must allocate space for additional storage. The physical unit of storage that the database server uses to allocate both the initial and subsequent storage space is called an *extent*. Figure 15-3 illustrates the concept of an extent.

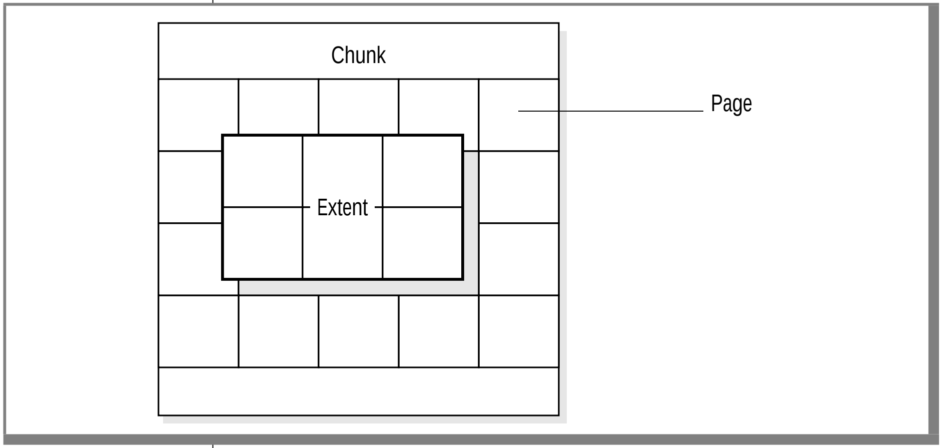


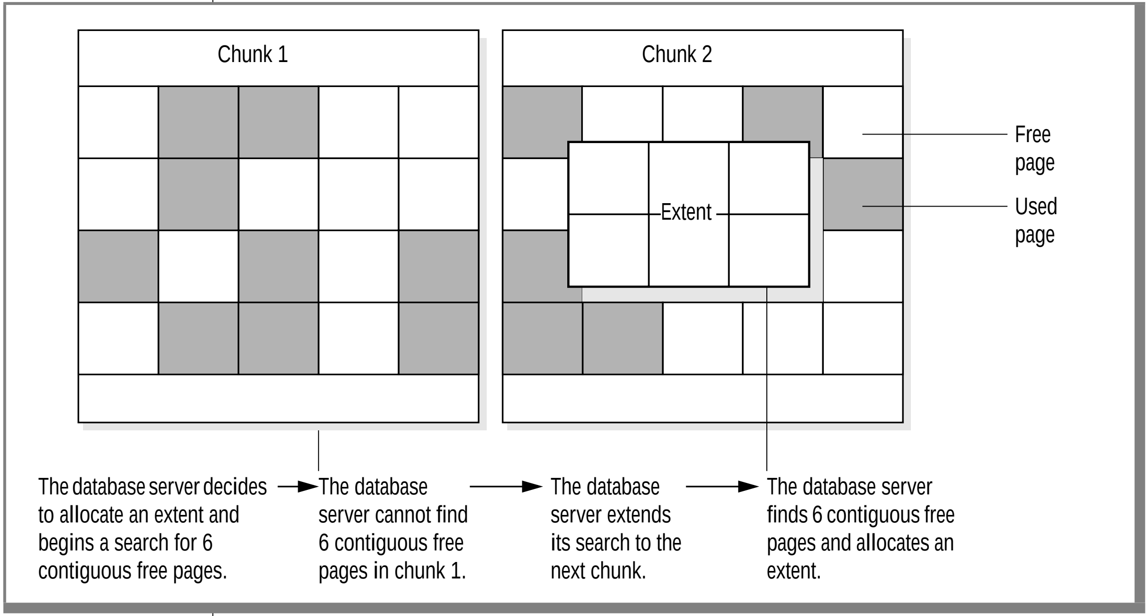
Figure 15-3
An Extent That Consists of Six Contiguous Pages on a Raw Disk Device

An extent consists of a collection of contiguous pages that store data for a given table. (See "Tables" on page 15-23.) Every permanent database table has two extent sizes associated with it. The *initial-extent* size is the number of kilobytes allocated to the table when it is first created. The *next-extent* size is the number of kilobytes allocated to the table when the initial extent (and any subsequent extents) becomes full. To specify the initial-extent size and next-extent size, use the CREATE TABLE and ALTER TABLE statements. For more information, see the *Informix Guide to SQL: Syntax*.

Figure 15-4 illustrates the following key concepts concerning extent allocation:

- An extent is always entirely contained in a chunk; an extent cannot cross chunk boundaries.
- If the database server cannot find the contiguous disk space that is specified for the next-extent size (six pages in Figure 15-4), it searches the next chunk in the dbspace for contiguous space.

Figure 15-4
Process of Extent Allocation


Chunk 1
Chunk 2
Extent
Free page
Used page
The database server decides to allocate an extent and begins a search for 6 contiguous free pages.
The database server cannot find 6 contiguous free pages in chunk 1.
The database server extends its search to the next chunk.
The database server finds 6 contiguous free pages and allocates an extent.

Disabling I/O Errors

Informix divides disabling I/O errors into two general categories: destructive and nondestructive. A disabling I/O error is destructive when the disk that contains a database becomes damaged in some way. This type of event threatens the integrity of data, and the database server marks the chunk and dbspace as down. The database server prohibits access to the damaged disk until you repair or replace the disk and perform a physical and logical restore.

A disabling I/O error is nondestructive when the error does not threaten the integrity of your data. Nondestructive errors occur when someone accidentally disconnects a cable, you somehow erase the symbolic link that you set up to point to a chunk, or a disk controller becomes damaged.

Before the database server considers an I/O error to be disabling, the error must meet two criteria. First, the error must occur when the database server attempts to perform an operation on a chunk that has at least one of the following characteristics:

- The chunk has no mirror.
- The primary or mirror companion of the chunk under question is off-line.

Second, the error must occur when the database server attempts unsuccessfully to perform one of the following operations:

- Seek, read, or write on a chunk
- Open a chunk
- Verify that chunk information on the first used page is valid

  The database server performs this verification as a sanity check immediately after it opens a chunk.

You can prevent the database server from marking a dbspace as down while you investigate disabling I/O errors. If you find that the problem is trivial, such as a loose cable, you can bring the database server off-line and then on-line again without restoring the affected dbspace from backup. If you find that the problem is more serious, such as a damaged disk, you can use **onmode -O** to mark the affected dbspace as down and continue processing.

Logical Units of Storage

The logical units of database server storage fall into the following categories:

- Units of logical storage that are dictated by relational database design:
  - Databases
  - Tables
- Units of logical storage that function as accounting entities:
  - Dbspaces
  - Tblspaces

A tblspace, for example, does not correspond to any particular part of a chunk or even to any particular chunk. The indexes and data that make up a tblspace might be scattered throughout your chunks. The tblspace, however, represents a convenient accounting entity for space across chunks devoted to a particular table. (See "Tables" on page 15-23.)

Multiple dbspaces managed as a single storage object are as follows:

- Dbslices
- Rootslices
- Logslices

The following sections describe these logical storage units.

Dbspaces

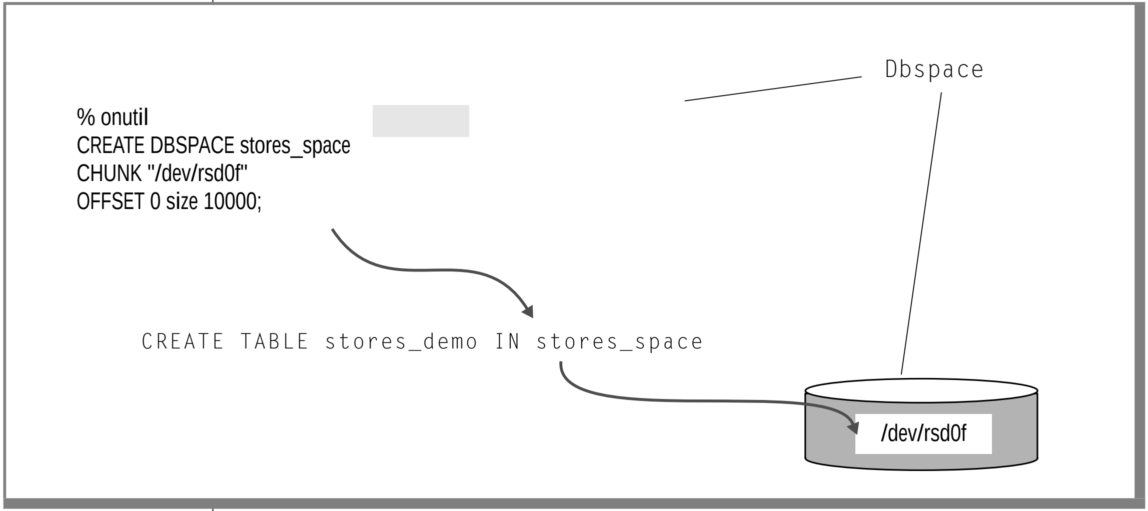
A key responsibility of the database server administrator is to control where the database server stores data. By storing high-use access tables or critical dbspaces (root dbspace, physical log, and logical log) on your fastest disk drive, you can improve performance. By storing critical data on separate physical devices, you ensure that when one of the disks holding noncritical data fails, the failure affects only the availability of data on that disk.

These strategies require the ability to control the location of data. The logical storage unit that provides this ability is the *dbspace*. The dbspace provides the critical link between the logical and physical units of storage. It allows you to associate physical units (such as chunks) with logical units (such as tables).

Control of Where Data Is Stored

As Figure 15-5 shows, to control the placement of databases or tables, you can use the IN *dbspace* option of the CREATE DATABASE or CREATE TABLE statements. (See "Tables" on page 15-23.)

Figure 15-5
Controlling Table Placement with the CREATE TABLE... IN Statement



```
% onutil
CREATE DBSPACE stores_space
CHUNK "/dev/rsd0f"
OFFSET 0 size 10000;
```

```
CREATE TABLE stores_demo IN stores_space
```

Before you create a database or table in a dbspace, you must first create the dbspace. For more information on how to create a dbspace, see "Creating a Dbspace" on page 16-13.

A dbspace includes one or more chunks, as Figure 15-6 on page 15-16 shows. You can add more chunks at any time. It is a high-priority task of a database server administrator to monitor dbspace chunks for fullness and to anticipate the need to allocate more chunks to a dbspace. (See "Monitoring the Database Server for Disabling I/O Errors" on page 16-32.) When a dbspace contains more than one chunk, you cannot specify the chunk in which the data resides.

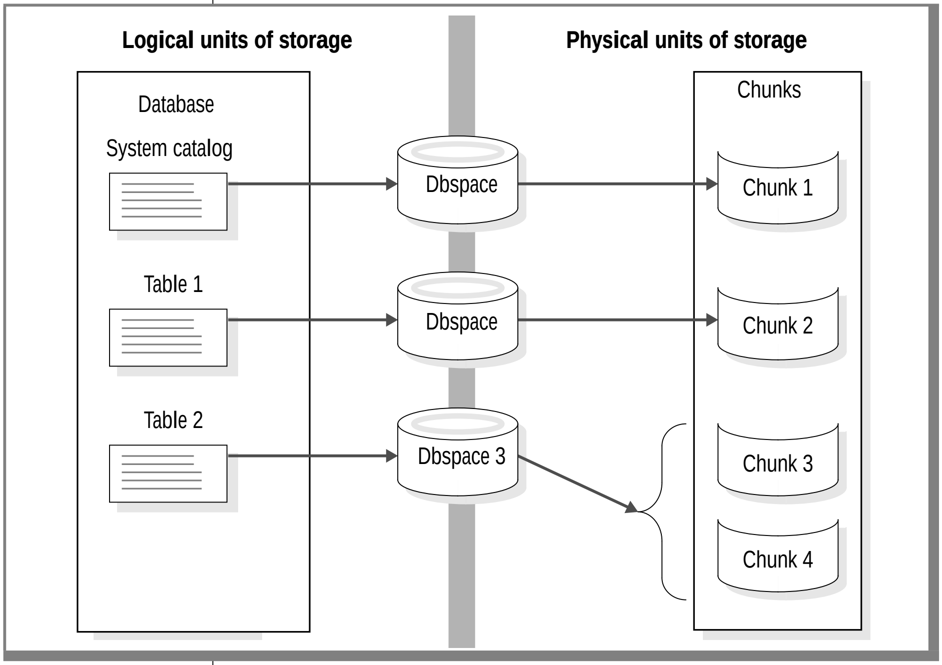


Figure 15-6
Dbspaces That Link Logical and Physical Units of Storage

The database server uses the dbspace to store databases and tables. (See "Tables" on page 15-23.)

You can mirror every chunk in a mirrored dbspace. As soon as the database server allocates a mirrored chunk, it flags all space in that mirrored chunk as full. See "Monitoring Disk Usage" on page 16-34.

You can use **onutil** to perform any of the following tasks related to dbspace management:

- Creating a dbspace (page 16-16)
- Creating a dbslice (page 16-16)
- Adding a chunk to a dbspace (page 16-20)
- Dropping a chunk from a dbspace (page 16-24)
- Dropping a dbspace (page 16-26)
- Dropping a dbslice (page 16-26)

Root Dbspace

The root dbspace is the initial dbspace that the database server creates. The root dbspace is special because it contains reserved pages and internal tables that describe and track all physical and logical units of storage. (For more information on these topics, see "Tables" on page 15-23 and the disk structures and storage chapter in the *Administrator's Reference*.) The initial chunk of the root dbspace and its mirror are the only chunks created during disk-space initialization. You can add other chunks to the root dbspace after disk-space initialization.

The following disk-configuration parameters in the ONCONFIG configuration file refer to the first (initial) chunk of the root dbspace:

- ROOTPATH
- ROOTOFFSET
- ROOTNAME
- MIRRORPATH
- MIRROROFFSET

The root dbspace is also the default dbspace location for any database created with the CREATE DATABASE statement.

The default value for the DBSPACETEMP configuration parameter in Extended Parallel Server is NOTCRITICAL. Therefore, implicit temporary tables do not use the root dbspace in Extended Parallel Server. For more information on temporary tables, refer to "Temporary Tables" on page 15-30.

"Size of the Root Dbspace" on page 15-36 explains how much space to allocate for the root dbspace. You can also add extra chunks to the root dbspace after you initialize database server disk space.

Temporary Dbspaces

A temporary dbspace is a dbspace reserved for the exclusive use of temporary tables. (See "Table Types" on page 15-25.)

The database server never drops a temporary dbspace unless it is explicitly directed to do so. A temporary dbspace is temporary only in the sense that the database server does not preserve any of the dbspace contents when the database server shuts down abnormally. Temporary dbspaces are designed exclusively for the storage of temporary tables.

Whenever you initialize the database server, all temporary dbspaces are reinitialized. The database server clears any tables that might be left over from the last time that the database server shut down.

The database server does not perform logical or physical logging for temporary dbspaces. Backup utilities do not include temporary dbspaces as part of a full-system dbspace backup. You cannot mirror a temporary dbspace.

For detailed instructions on how to create a temporary dbspace, see "Creating a Temporary Dbspace" on page 16-15.

For more information on temporary dbspaces on Extended Parallel Server, see dbspaces for temporary tables in your *Performance Guide*.

Advantages of Using Temporary Dbspaces

The database server logs table creation, the allocation of extents, and the dropping of the table for a temporary table in a standard dbspace. In contrast, the database server suppresses all logical logging for implicit temporary tables and explicit temporary tables created with the WITH NO LOG options that reside in a temporary dbspace. Logical-log suppression in temporary dbspaces reduces the number of log records to roll forward during logical recovery as well, thus improving the performance during critical down time.

The database server does not perform any physical logging in temporary dbspaces. This practice helps performance in two ways. First, physical logging itself generates I/O. Reducing I/O always improves performance. Second, whenever the physical log becomes 75 percent full, a checkpoint occurs. Checkpoints require a brief period of inactivity to complete, which can have a negative impact on performance. When temporary tables reside in temporary dbspaces, the database server does not perform physical logging for operations on the temporary tables, thus requiring fewer checkpoints.

Using temporary dbspaces to store temporary tables also reduces the size of your dbspace backup because the database server does not backup temporary dbspaces.

Dbslices

Informix recommends that you partition VLDBs across many coservers in Extended Parallel Server. Each table fragment is stored in its own dbspace; therefore, you can create thousands of storage objects (fragments, dbspaces, chunks, and so on) spread across multiple coservers. Managing these individual storage objects can be complex and error-prone unless you can manage groups of them.

Extended Parallel Server uses a *dbslice* to manage many storage objects. A *dbslice* is a named set of dbspaces that you can manage as a single storage object. A dbslice contains all of the traditional database server storage units: dbspaces, chunks, and so on.

A dbslice facilitates management of storage objects because you can refer to all of the storage objects for a single table with a single name, the dbslice name. For example, to fragment a table across 100 coservers, you can use the following CREATE TABLE statement to specify a single dbslice name instead of 100 dbspace names:

```
CREATE TABLE customer
    (cust_id integer,
    ...
    )
BY HASH (cust_id)
        IN customer_dbslc;
```

In this example, the SQL operation takes place for all of the underlying dbspaces in the **customer_dbslc** dbslice.

To define a dbslice, use the **onutil** CREATE DBSLICE command. The **onutil** command-line utility accepts commands that create, alter, and drop storage objects (dbslices and cogroups).

When you create a dbslice, you specify the cogroup name so that Extended Parallel Server knows the coservers on which to create dbspaces. For example, you might create a dbslice from an accounting cogroup. The following partial commands show how to create a cogroup and dbslice:

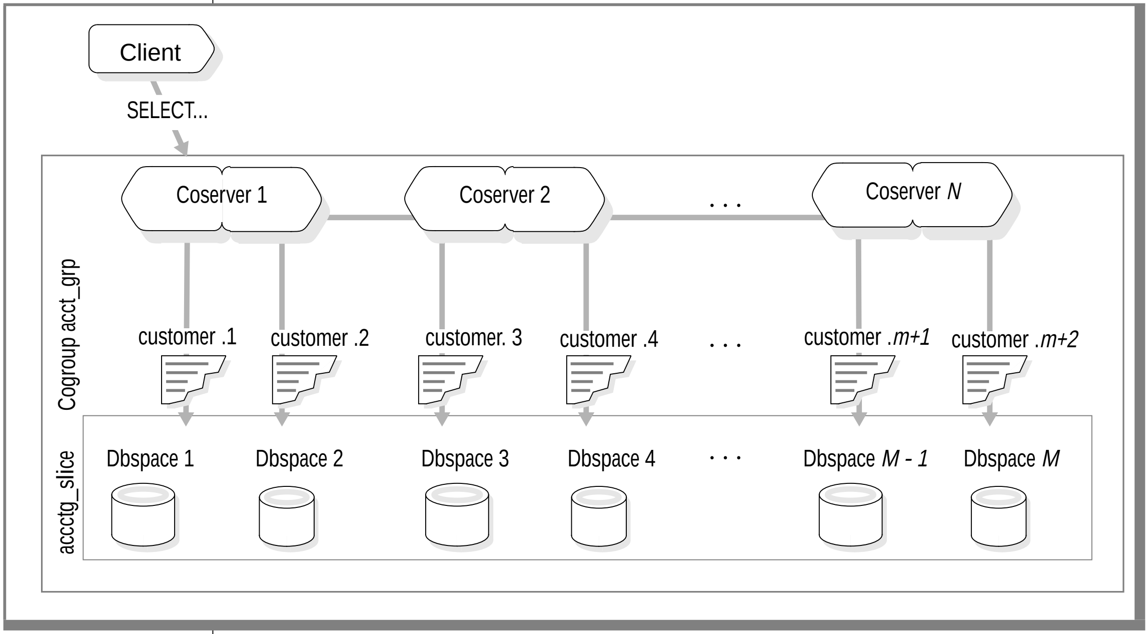
```
% onutil
1> create cogroup acctg_group
2>   .
3>   .
4>   .
5> create dbslice acctg_dbslc
6>   FROM acctg_group . . .
```

You do not need to specify the names explicitly for all of the individual dbspaces that are associated with the partitioned tables. Extended Parallel Server generates the dbspace names for you.

Tip: *To add dbspaces to a dbslice, use the **onutil** ALTER DBSLICE command.*

For more details on the **onutil** CREATE COGROUP, **onutil** CREATE DBSLICE, and **onutil** ALTER DBSLICE commands, see the utilities chapter of the *Administrator's Reference*. Figure 15-7 illustrates a cogroup and a dbslice.

Figure 15-7
A Cogroup and Dbslice



Rootslices

A *rootslice* is a named set of root dbspaces that you can manage as a single storage object. Extended Parallel Server creates a root dbspace on each coserver. To facilitate management of multiple root dbspaces, Extended Parallel Server provides rootslices.

Temporary Dbslices

A temporary dbslice is a named set of temporary dbspaces that reside on multiple coservers. You can manage temporary dbspaces as a single storage object by using the dbslice name.

For information on how to create a temporary dbslice, refer to the TEMP keyword in the **onutil** CREATE DBSLICE command in the *Administrator's Reference*.

Databases

A database is a logical storage unit that contains tables and indexes. (See "Tables" on page 15-23.) Each database also contains a system catalog that tracks information about many of the elements in the database, including tables, indexes, SPL routines, and integrity constraints.

A database resides in the dbspace specified by the CREATE DATABASE statement. When you do not explicitly name a dbspace in the CREATE DATABASE statement, the database resides in the root dbspace. When you *do* specify a dbspace in the CREATE DATABASE statement, this dbspace is the location for the following tables:

- Database system catalog tables
- Any table that belongs to the database

Figure 15-8 shows the tables contained in the **stores_demo** database.

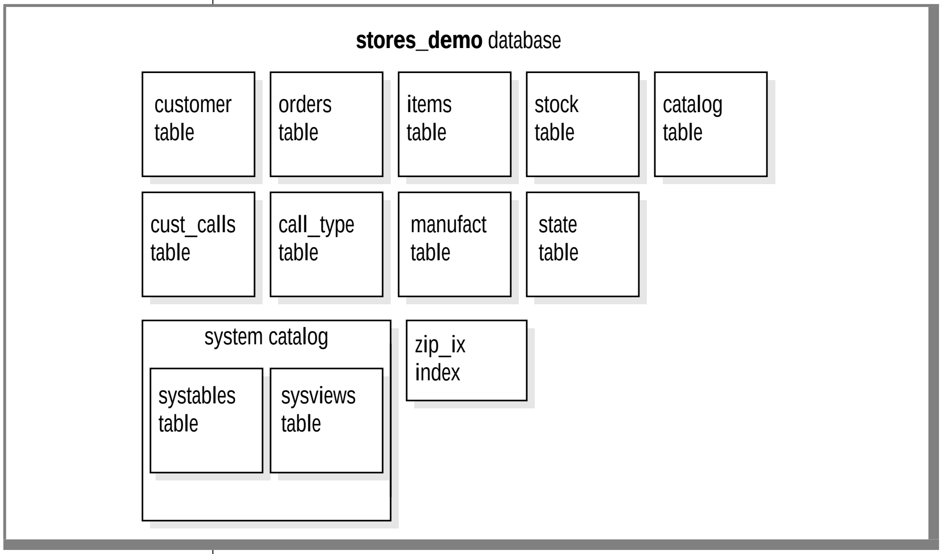


Figure 15-8
The stores_demo Database

The size limits that apply to databases are related to their location in a dbspace. To be certain that all tables in a database are created on a specific physical device, assign only one chunk to the device, and create a dbspace that contains only that chunk. Place your database in that dbspace. When you place a database in a chunk assigned to a specific physical device, the database size is limited to the size of that chunk.

For instructions on how to list the databases that you create, see "Displaying Databases" on page 16-32.

Tables

In relational database systems, a table is a row of column headings together with zero or more rows of data values. The row of column headings identifies one or more columns and a data type for each column.

When users create a table, the database server allocates disk space for the table in a block of pages called an extent. (See "Extents" on page 15-11.) You can specify the size of both the first and any subsequent extents.

Users can place the table in a specific dbspace by naming the dbspace when they create the table (usually with the IN *dbspace* option of CREATE TABLE). When the user does not specify the dbspace, the database server places the table in the dbspace where the database resides.

Users can also fragment a table over more than one dbspace. Users must define a distribution scheme for the table that specifies which table rows are located in which dbspaces.

Users can also fragment a table over more than one dbspace. Users must define a distribution scheme for the table that specifies which table rows are located in which dbspaces. For more information about distribution schemes, see the *Informix Guide to Database Design and Implementation*.

A table or table fragment resides completely in the dbspace in which it was created. The database server administrator can use this fact to limit the growth of a table by placing a table in a dbspace and then refusing to add a chunk to the dbspace when it becomes full.

A table, composed of extents, can span multiple chunks, as Figure 15-9 shows.

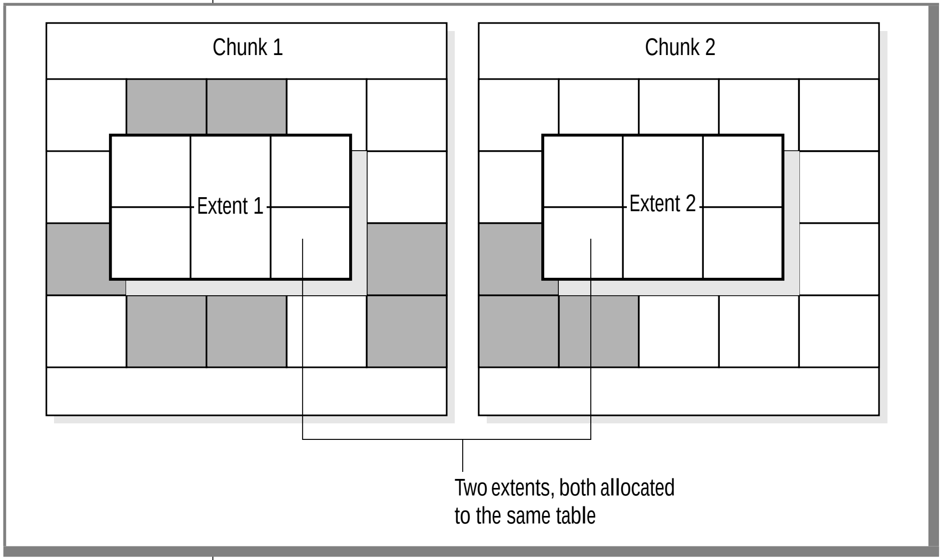


Figure 15-9
Table That Spans More than One Chunk

For advice on where to store your tables, see "Disk-Layout Guidelines" on page 15-39 and your *Performance Guide*.

Table Types

Figure 15-10 lists the properties of the six types of tables available with Extended Parallel Server. The flag values are the octal values for each table type in the flags column of **systables**.

Figure 15-10
Table Characteristics

Type	Permanent	Logged	Indexes	Light Append Used	Rollback Available	Recoverable	Restorable from Backup	Loading Mode	Flag Value (0x0000-)
SCRATCH	No	No	No	Yes	No	No	No	Either	40000
TEMP	No	Yes	Yes	Yes	Yes	No	No	Either	10000
RAW	Yes	No	No	Yes	No	Depends	Depends	Either	1000
STATIC	Yes	No	Yes	No	No	Depends	Depends	None	2000
OPERATIONAL	Yes	Yes	Yes	Yes	Yes	Yes	Depends	Either	4000
STANDARD	Yes	Yes	Yes	No	Yes	Yes	Yes	Deluxe	8000
EXTERNAL	Yes	No	No	Yes	No	No	See 15-28	Either	20000

For information about logging, see "Logging and Nonlogging Tables" on page 18-8. For information about fast recovery, see "Fast Recovery of Tables" on page 24-23. For information on restoring various table types, see the *Backup and Restore Guide*.

Tip: *"Depends" in Figure 15-10 means that a table is recoverable or restorable only if it has not been updated.*

Scratch and Temp Tables

Scratch and temp tables are temporary tables that are dropped when the user session closes, the database server shuts down, or on reboot after a failure. You cannot recover, back up, or restore temp and scratch tables.

Scratch and temp tables support bulk operations such as light appends, which add rows quickly to the end of each table fragment. For more information on light appends, refer to your *Performance Guide*.

Scratch tables are nonlogging temporary tables that do not support indexes, constraints, or rollback.

Temp tables are logged tables by default, and they support indexes, constraints, and rollback.

Extended Parallel Server creates explicit temporary tables according to the following criteria:

- If the query used to populate the TEMP table produces no rows, the database server creates an empty, unfragmented table.
- If the rows that the query produces do not exceed 8 kilobytes, the temporary table resides in only one dbspace.
- If the rows exceed 8 kilobytes, Extended Parallel Server creates multiple fragments and uses a round-robin fragmentation scheme to populate them.

SELECT...INTO TEMP or SELECT...INTO SCRATCH statements operate in parallel across coservers, just as ordinary inserts do. Extended Parallel Server automatically supports fragmented temporary tables across nodes when those tables are explicitly created with SELECT...INTO TEMP or SELECT...INTO SCRATCH.

Important: *A TEMP type table is a logging table by default. If you want to use the temporary dbspaces or dbslices specified in the DBSPACETEMP configuration parameter or* ***DBSPACETEMP*** *environment variable, you must specify the WITH NO LOG clause when you use the SELECT...INTO TEMP statement.*

Tip: *A SCRATCH table is nonlogging by default. When you execute the SELECT...INTO SCRATCH statement, the database server uses the temporary dbspaces or dbslices specified in the DBSPACETEMP configuration parameter or* ***DBSPACETEMP*** *environment variable.*

Raw Permanent Tables

Raw tables are nonlogging permanent tables that use light appends. You can use the express loading mode to load them.

Updates, inserts, and deletes are supported but not logged. Raw tables do not support indexes, referential constraints, or rollback. You can restore a raw table from the last physical backup if it has not been updated since then. You can recover a raw table if it has not been updated since the previous checkpoint.

Raw tables are intended for the initial loading and validation of data. Once you have completed these steps, you should alter the table to a higher level. If an error or failure occurs during loading of a raw table, the resulting data is whatever was on the disk at the time of the failure.

Static Permanent Tables

Static tables are nonlogging read-only permanent tables. They do not support inserts, updates, and deletes. However, you can create and drop nonclustered indexes and referential constraints because they do not affect the data. Their advantage is that the server can use light scans and avoid locking during the execution of queries because static tables are read-only.

Static tables do not support rollback. Static tables inherit the recovery characteristics of the tables they were created from. If you alter a raw table to static table, you will be able to recover or restore it if it was not updated since the previous checkpoint or backup. If you alter a standard table to a static table, you always will be able to recover or restore it.

Operational Permanent Tables

Operational tables are logging permanent tables that allow indexes and constraints and fast update operations. They allow light appends only if the table contains no indexes or constraints.

If an operational table has indexes, the database server uses deluxe mode to load it. They perform row-by-row logging of insert, update, and delete operations but do not log light appends. If an operational table does not have indexes, the database server uses express mode to load it.

You can roll back operations or recover after a crash with operational tables. You can restore an operational table unless a light append occurred since the most recent backup. Operational tables are intended for use in situations when the data is derived from another source, so restorability is not an issue, but when rollback and recoverability are required.

Standard Permanent Tables

A standard table is the same as a table in a logged database that the database server creates. All operations are logged, record by record, so standard tables can be restored from a backup and support recoverability and rollback.

Standard tables do not use light appends, so you must use the deluxe loading mode. You must load standard tables record-by-record with every operation logged.

External Tables

An external table is a data file that you use to load and unload data. The database server performs express-mode and deluxe-mode loads. Use the CREATE EXTERNAL TABLE statement to load data into an external table.

Because external tables are outside of the database server, you cannot recover, roll back, or use ON-Bar to restore them. However, you can back up the external tables with a file backup program.

You can use express mode to load a raw or operational table without any indexes to an external table. You can use deluxe mode to load a raw or operational table with indexes, or a standard table to an external table. However, you cannot load a static table to an external table. For more information, refer to the chapter on loading with external tables in the *Administrator's Reference*.

Rollback of Operational and Raw Tables

The following examples show how you can roll back transactions for operational tables but not for raw tables.

Rollback of Operational Tables

After you roll back the transaction on **tab_op**, the SELECT command shows that the inserted row was rolled back.

```
CREATE OPERATIONAL TABLE tab_op (c1 int); # create op table
BEGIN WORK; #start transaction
INSERT INTO tab_op values (1); # insert a row into table
ROLLBACK WORK; # transaction rolled back
SELECT * FROM tab_op; # inserted row is gone
```

Rollback of RawTables

After you roll back the transaction on **tab_raw**, the SELECT command shows that the inserted row is not rolled back.

```
CREATE RAW TABLE tab_raw (c1 int); # create raw table
BEGIN WORK; #start transaction
INSERT INTO tab_raw values (2); # insert a row into table
ROLLBACK WORK; # transaction rolled back
SELECT * FROM tab_raw; # inserted row remains
```

Switching Between Table Types

Use the ALTER TABLE command to switch between types of permanent tables. If the table does not meet the restrictions of the new type, the alter fails and produces an explanatory error message. The following restrictions apply to table alteration:

- You must drop indexes and referential constraints before you alter a table to type RAW.
- You must perform a level-0 backup before you alter a table to type STANDARD.
- You cannot alter a TEMP or SCRATCH temporary table.

Temporary Tables

The two types of temporary tables are *explicit* temporary tables and *implicit* temporary tables. You can create temporary tables in a standard dbspace or temporary dbspace.

An *explicit* temporary table is a temporary table that you create with one of the following SQL statements:

- TEMP TABLE option of the CREATE TABLE statement
- INTO TEMP clause of the SELECT statement

  ```
  SELECT * FROM customer INTO TEMP temp_table
  ```
- SCRATCH TABLE option of the CREATE TABLE statement
- INTO SCRATCH clause of the SELECT statement

When an application creates an explicit temporary table, it exists until the application takes one of the following actions:

- The application terminates.
- The application closes the database in which the table was created and opens a database in a different database server.
- The application closes the database in which the table was created.

When any of these three events occurs, the database server deletes the temporary table.

An *implicit* temporary table is a temporary table or file that the database server creates as part of processing.

The following statements might require temporary disk space:

- Statements that include a GROUP BY or ORDER BY clause
- Statements that use aggregate functions with the UNIQUE or DISTINCT keywords
- SELECT statements that use auto-index or hash joins
- Complex CREATE VIEW statements
- DECLARE statements that create a scroll cursor
- Statements that contain correlated subqueries
- Statements that contain subqueries that occur within an IN or ANY clause

- CREATE INDEX statements
- DECLARE statements that use the SCROLL CURSOR option

The database server deletes an implicit temporary table when the processing that initiated the creation of the table is complete.

If the database server shuts down without adequate time to clean up temporary tables, it performs temporary table cleanup as part of the next initialization. (To request shared-memory initialization without temporary table cleanup, execute **oninit** with the **-p** option.)

Important: *You cannot create a temporary external table on Extended Parallel Server.*

Storage of Temporary Tables

Adequate temporary space in the appropriate dbspaces to store temporary tables and files is critical to the overall performance of your database server. The output file of the SQL statement SET EXPLAIN ON lists temporary-file requirements.

The dbspace in which the database server stores temporary tables depends on whether the table is an explicit or implicit table. The following sections examine both cases in detail.

Explicit Temporary Tables

When you create an explicit temporary table using the IN *dbspace* option of CREATE TEMP TABLE, the database server stores the temporary table in that dbspace.

When you do not use the IN *dbspace* option of CREATE TEMP TABLE, or when you create the explicit table with SELECT... INTO TEMP, the database server checks the **DBSPACETEMP** environment variable and the DBSPACETEMP configuration parameter. (The environment variable supersedes the configuration parameter.) When **DBSPACETEMP** is set, the database server stores the explicit temporary table in one of the dbspaces specified in the list.

The database server keeps track of the last dbspace in the list that it used to store a temporary table. When the database server receives another request for temporary storage space, it uses the next dbspace in the list. In this way, the database server spreads I/O evenly across the temporary storage space that you specify in **DBSPACETEMP**.

Important: *In the case of a database with logging, you must include the WITH NO LOG clause in the SELECT... INTO TEMP statement to place the explicit temporary tables in the dbspaces listed in the DBSPACETEMP configuration parameter or the* ***DBSPACETEMP*** *environment variable.*

When you do not specify any temporary dbspaces in **DBSPACETEMP**, or the temporary dbspaces that you specify have insufficient space, the database server creates the table in a standard (nontemporary) dbspace according to the following rules:

- If you created the temporary table with CREATE TEMP TABLE, the database server stores this table in the dbspace that contains the database to which the table belongs.
- The default value for the DBSPACETEMP configuration parameter in Extended Parallel Server is NOTCRITICAL. Therefore, explicit temporary tables do not use the root dbspace in Extended Parallel Server. For more information on temporary tables, refer to "Temporary Tables" on page 15-30.

Flexible Temporary Tables

A *flexible (flex) temporary table* is an explicit temporary table that the database server creates and then fragments (round-robin method) automatically. The following query, for example, creates a flex temporary table:

```
SELECT * FROM customer INTO SCRATCH temp_table
```

One advantage of a flex temporary table over a table created with CREATE TABLE syntax is that you do not need to know column names and data types.

Extended Parallel Server uses an SQL operator to optimize use of dbspaces and dbslices for temporary storage. When data is received, a fragment of that table is created in one of the available dbspaces (as determined by the value of DBSPACETEMP), and data is light-appended to the fragment. If the dbspace that is being used becomes full, the SQL operator attempts to write incoming data into another dbspace on the same node. For more information on fragmenting a flex temporary table, refer to the *Performance Guide*.

Important: *A coserver can use and access only its own dbspaces for temporary space. Although temporary tables can be deliberately fragmented across dbspaces in the same way as permanent tables, a coserver inserts data only into the fragments that it manages.*

Implicit Temporary Tables

The database server stores implicit temporary tables in one of the dbspaces that you specify in the **DBSPACETEMP** environment variable or the DBSPACETEMP configuration parameter. The environment variable supersedes the configuration parameter.

When the **DBSPACETEMP** environment variable and the DBSPACETEMP configuration parameter are not set, Extended Parallel Server stores the temporary table in the default dbspaces or dbslices specified in the DBSPACETEMP configuration parameter. The default value of DBSPACETEMP in **onconfig.std** or **onconfig.xps** is NOTCRITICAL, which includes all standard dbspaces or dbslices except the root or logs.

For information on how to create temporary dbspaces, refer to "Creating a Temporary Dbspace" on page 16-15.

Tblspaces

Database server administrators sometimes need to track disk use by a particular table. A *tblspace* contains all the disk space allocated to a given table or table fragment (if the table is fragmented). A separate tblspace contains the disk space allocated for the associated index.

The table tblspace contains the following types of pages:

- Pages allocated to data
- Pages allocated to indexes
- Pages used to store TEXT or BYTE data in the dbspace
- Bit-map pages that track page use within the table extents

The index tblspace contains the following types of pages:

- Pages allocated to indexes
- Bit-map pages that track page use within the index extents

illustrates the tblspaces for three tables that form part of the **stores_demo** database. Only one table (or table fragment) exists per tblspace. An index resides in a separate tblspace from the associated table. Blobpages represent TEXT or BYTE data stored in a dbspace.

Extent Interleaving

The database server allocates the pages that belong to a tblspace as extents. Although the pages within an extent are contiguous, extents might be scattered throughout the dbspace where the table resides (even on different chunks). Figure 15-11 depicts this situation with two noncontiguous extents that belong to the tblspace for **table_1** and a third extent that belongs to the tblspace for **table_2**. A **table_2** extent is positioned between the first **table_1** extent and the second **table_1** extent. When this situation occurs, the extents are interleaved. Because sequential access searches across **table_1** require the disk head to seek across the **table_2** extent, performance is worse than if the **table_1** extents were contiguous. For instructions on how to avoid and eliminate interleaving extents, see your *Performance Guide*.

Figure 15-11
Three Extents That Belong to Two Different Tblspaces in a Single Dbspace

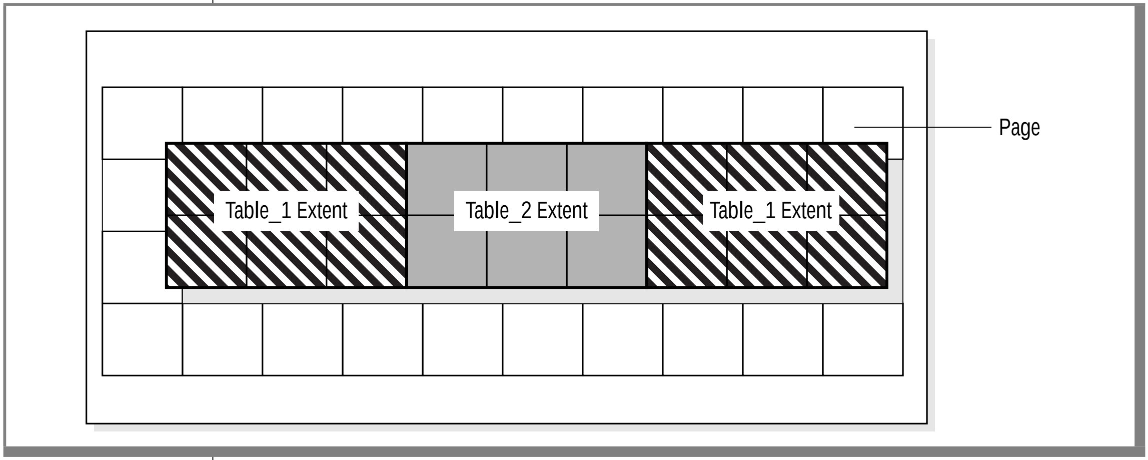


Table Fragmentation and Data Storage

The fragmentation feature gives you additional control over where the database stores data. You are not limited to specifying the locations of individual tables and indexes. You can also specify the location of table and index *fragments*, which are different parts of a table or index that reside on different storage spaces. You can fragment the following storage spaces:

- Dbspaces
- Dbslices

For more information about fragmentation, see "Fragmentation" on page 17-3.

Amount of Disk Space Needed to Store Data

To determine how much disk space you need, follow these steps:

1. Calculate the size requirements of the root dbspace.
2. Estimate the total amount of disk space to allocate to all the database server databases, including space for overhead and growth.

The following sections explain these steps.

Size of the Root Dbspace

To calculate the size of the root dbspace, take the following storage structures into account:

- The physical- and logical-log files
- Temporary tables
- Data
- Control information
- The safewrite area

You need not store the physical log, the logical log, or the temporary tables in the root dbspace. Include calculations for these items only if you plan to continue to store them in the root dbspace.

If you plan to move the physical and logical logs, the initial configuration for the root dbspace might differ markedly from the final configuration. You can resize the root dbspace after you remove the physical and logical logs. However, the root dbspace must be large enough for the minimum size configuration during disk initialization.

The sections that follow discuss each storage structure in the root dbspace.

Physical and Logical Logs

The value stored in the ONCONFIG parameter PHYSFILE defines the size of your physical log. Advice on sizing your physical log is contained in "Size and Location of the Physical Log" on page 22-5.

To calculate the size of the logical-log files, multiply the value of the ONCONFIG parameter LOGSIZE by the number of logical-log files. For advice on sizing your logical log, see "Size and Number of Logical-Log Files" on page 20-7.

Temporary Tables

Analyze end-user applications to estimate the amount of disk space that the database server might require for implicit temporary tables. "Temporary Tables" on page 15-30 contains a list of statements that require temporary space. Try to estimate how many of these statements are to run concurrently. The space occupied by the rows and columns that are returned provides a good basis for estimating the amount of space required.

The database server creates implicit temporary files when you perform a warm restore. The largest implicit temporary file that the database server creates during a warm restore is equal to the size of your logical log. You calculate the size of your logical log by multiplying the value of LOGSIZE by LOGFILES. For more information on these configuration parameters, see "Size and Number of Logical-Log Files" on page 20-7.

You must also analyze end-user applications to estimate the amount of disk space that the database server might require for explicit temporary tables. See "Temporary Tables" on page 15-30.

By default, the database server stores both implicit and explicit temporary tables in the root dbspace. However, if you decide not to store your temporary tables in the root dbspace, you can use the **DBSPACETEMP** environment variable and configuration parameter to specify a list of dbspaces for temporary files and tables. See "Storage of Temporary Tables" on page 15-31.

Critical Data

Next, decide if users store databases or tables in the root dbspace. If the root dbspace is the only dbspace that you intend to mirror, place all critical data there for protection. Otherwise, store databases and tables in another dbspace.

Estimate the amount of disk space, if any, that you need to allocate for tables stored in the root dbspace.

Control Information

The total amount of disk space required for the database server control information is 3 percent of the size of the root dbspace (sum of physical and logical log, temporary space, and data) plus 25 pages, expressed as kilobytes (or 25 times the database server page size).

Safewrite Area

Each coserver stores information about the current coserver-configuration in a portion of the root dbspace called the safewrite area. Storing this data in the root dbspace ensures data consistency across coservers in the event of a failure. You can use the CONFIGSIZE, MAX_CHUNKS, MAX_DBSPACES, and MAX_DBSLICES configuration parameters to specify the size of the safewrite area. For more information about these parameters, refer to the chapter on configuration parameters in the *Administrator's Reference*.

If you run out of space for the safewrite area, the database server writes a message to the transaction log. To increase the amount of space in the safewrite area, add a chunk to the root dbspace before you restart the database server.

Amount of Space That Databases Require

The amount of additional disk space required for the database server data storage depends on the needs of your end users, plus overhead and growth. Every application that your end users run has different storage requirements. The following list suggests some of the steps that you can take to calculate the amount of disk space to allocate (beyond the root dbspace):

1. Decide how many databases and tables you need to store. Calculate the amount of space required for each one.
2. Calculate a growth rate for each table and assign some amount of disk space to each table to accommodate growth.
3. Decide which databases and tables you want to mirror.

For instructions about calculating the size of your tables, refer to your *Performance Guide*.

Disk-Layout Guidelines

The following are typical goals for efficient disk layout:

- Limiting disk-head movement
- Reducing disk contention
- Balancing the load
- Maximizing availability

You must make some trade-offs between these goals when you design your disk layout. For example, separating the system catalog tables, the logical log, and the physical log can help reduce contention for these resources. However, this action can also increase the chances that you have to perform a system restore.

The sections that follow discuss various strategies for meeting disk-layout goals.

Dbspace and Chunk Guidelines

This section lists some general strategies for disk layout that do not require any information about the characteristics of a particular database.

Strive to Associate Partitions with Chunks

When you allocate disk space (buffered or unbuffered files), you allocate it in chunks. A dbspace is associated with one or more chunks. You must allocate at least one chunk for the root dbspace.

Informix recommends that you format your disks so that each chunk is associated with its own disk partition. You can easily track disk-space use when you define every chunk as a separate partition (or device). You can also avoid errors caused by miscalculated offsets.

A disk that is already partitioned might require the use of offsets. For details, see "Allocating Raw Disk Space on UNIX" on page 16-10.

Mirror Critical Data Dbspaces

Mirror the critical dbspaces: the root dbspace, the dbspace that contains the physical log, and the dbspace that contains the logical-log files. You specify mirroring on a chunk-by-chunk basis. Locate the primary and the mirrored chunk on different disks. Ideally, different controllers handle the different disks. Figure 15-12 shows a primary chunk and its mirror.

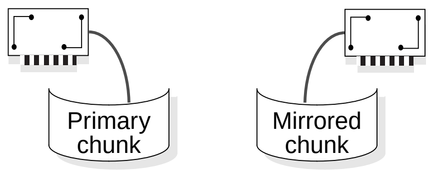


Figure 15-12
Ideal Disk Layout for Primary Chunk and Associated Mirrored Chunk

Spread Temporary Storage Space Across Multiple Disks

You can use the **DBSPACETEMP** environment variable and configuration parameter to store a list of dbspaces used for temporary storage. The list can include both temporary and standard dbspaces. To achieve load balancing, design the list so that your temporary disk space is spread across multiple disks. For instructions on how to set DBSPACETEMP, see the chapter on configuration parameters in the *Administrator's Reference*.

Move the Logical and Physical Logs from the Root Dbspace

Whether or not you use logging tables, the logical log and physical log both contain data that the database server accesses frequently. Reserved pages are also accessed frequently; they contain internal tables that describe and track all dbspaces, chunks, databases, and tblspaces.

When you allocate disk space (buffered or unbuffered files), you allocate it in chunks. A dbspace is associated with one or more chunks. You must allocate at least one chunk for the root dbspace.

By default, the database server stores the logical and physical logs together with the reserved pages in the root dbspace. Storing the logical and physical logs together is convenient if you have a small, low-volume transaction-processing system. However, maintaining these files together in the root dbspace can become a source of contention as your database system grows.

To reduce this contention and provide better load balancing, move the logical and physical logs to separate partitions or, even better, separate disk drives. For optimum performance, consider creating two additional dbspaces: one for the physical log and one for the logical log. When you move the logs, avoid storing them in a dbspace that contains high-use tables. Instead, consider storing them in a dbspace dedicated to storing only the physical or logical log. For more advice on where to store your logs, see "Location of the Physical Log" on page 22-8 and "Location of Logical-Log Files" on page 20-9.

For instructions on how to change the location of the logical and physical log, see "Using a Text Editor to Change Physical-Log Location or Size" on page 23-5 and "Moving a Logical-Log File to Another Dbspace" on page 21-7.

Consider Account Backup-and-Restore Performance

When you plan your disk layout, consider how the configuration that you choose affects your backup-and-restore procedure. This section describes two configurations that can have a significant impact on your backup-and-restore procedure.

Cluster Catalogs with the Data That They Track

When a disk that contains the system catalog of a particular database fails, the entire database remains inaccessible until you restore the system catalog. Informix recommends that you do not cluster the system catalog tables for all databases in a single dbspace but instead place the catalogs with the data that they track.

Reconsider Separating the Physical and Logical Logs

Although it makes sense from a performance perspective to separate the root dbspace from the physical and logical logs, and the two logs from one another, this configuration is the least desirable in terms of recovery.

Whenever a disk that contains critical information (the root dbspace, physical log, and logical log) fails, the database server comes off-line. In addition, the database server administrator must restore all the database server data, starting in off-line mode, from a level-0 backup before processing can continue.

When you separate the root dbspace from the physical- and logical-log files, you increase the probability that, if a disk fails, it is one that contains critical information (either the root dbspace, physical log, or logical log). For information on how to fragment to improve backup and restore characteristics, see your *Performance Guide*.

Table-Location Guidelines

This section lists some strategies for optimizing the disk layout, given certain characteristics about the tables in a database. You can implement many of these strategies with a higher degree of control using table fragmentation. For a discussion of how to optimize your disk layout using table fragmentation, refer to your *Performance Guide*.

Isolate High-Use Tables

You can place a table with high I/O activity on a disk device dedicated to its use and thus reduce contention for the data stored in the table. When disk drives have different performance levels, you can put the tables with the highest frequency of use on the fastest drives. Placing two high-use tables on separate disk devices reduces competition for disk access when joins are formed between the two tables or when the two tables experience frequent, simultaneous access from multiple applications.

To isolate a high-use table on its own disk device, assign the device to a chunk, and assign the same chunk to a dbspace. Finally, place the frequently used table in the dbspace just created using the IN *dbspace* option of CREATE TABLE. Figure 15-13 on page 15-44 illustrates this strategy by showing optimal placement of three frequently used tables.

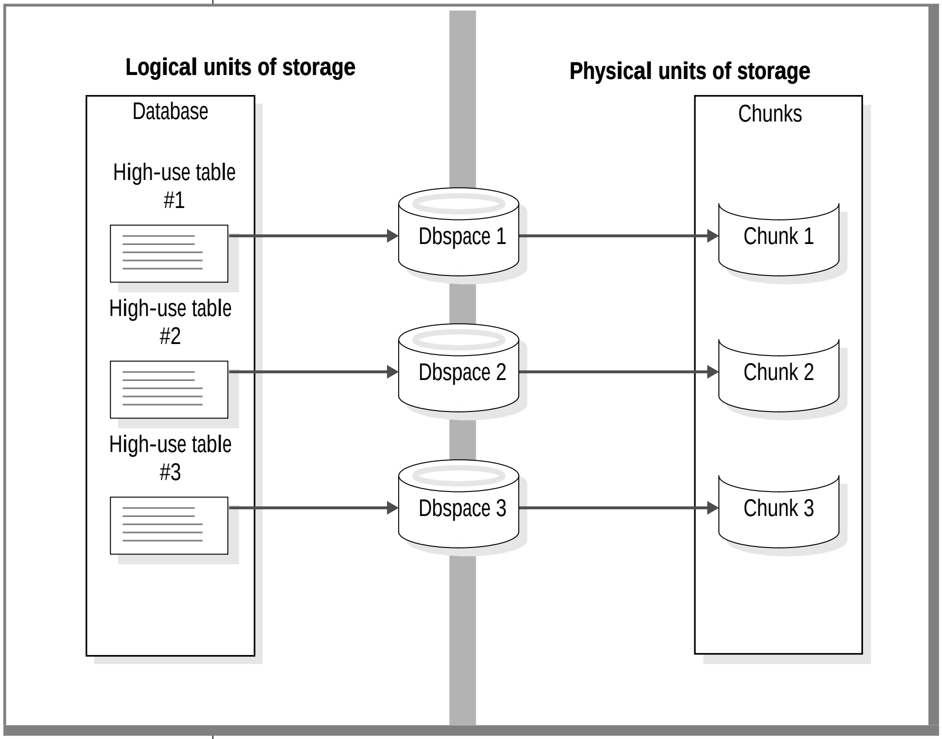


Figure 15-13
Example of High-Use Table Isolation

To take this strategy a step further, fragment a high-use table over multiple disk devices. If you choose an appropriate distribution scheme, the database server routes queries to the appropriate fragment, thereby reducing contention on any single fragment. For more information, see your *Performance Guide*.

If you have doubts whether spreading your tables across multiple disks can improve performance for your particular configuration, run the **-g iof** option of **onstat**. This option displays the level of I/O operations against each chunk. For details about **onstat**, see the utilities chapter in the *Administrator's Reference*.

Consider Mirroring

You can mirror critical tables and databases to maximize availability. You specify mirroring on a chunk-by-chunk basis. Locate the primary and mirrored chunks for critical tables on different disks. Ideally, different controllers handle the different disks.

Fragmentation gives you a higher level of control over this process. That is, you can mirror chunks that contain specific table fragments. For more information, see your *Performance Guide*.

Group Tables with Backup and Restore in Mind

When you decide where to place your tables, keep in mind that if a device containing a dbspace fails, all tables in that dbspace are inaccessible. However, tables in other dbspaces remain accessible. The accessibility (or inaccessibility) of dbspace might influence which tables you group together in a particular dbspace.

Although you must perform a cold restore if a dbspace that contains critical information fails, you need only perform a warm restore if a noncritical dbspace fails. This situation might influence which dbspace you use to store critical information. If you use ON-Bar for your backup and restore tool, refer to your *Backup and Restore Guide* for more information. If you use ON-Archive for your backup and restore tool, refer to your *Archive and Backup Guide* for more information.

Fragmentation gives you greater granularity of backup and restore. When you fragment a table, you can still access the fragments located in the other dbspaces in the event of a dbspace failure. For more information, see your *Performance Guide*.

Place High-Use Tables on Middle Partition of Disk

To minimize disk-head movement, place the most-frequently accessed data in partitions as close to the middle of the disk as possible. See Figure 15-14. When a disk device is partitioned, the central partitions generally experience the fastest access time. Place the least-frequently used data on the outermost or innermost partitions. This overall strategy minimizes disk-head movement.

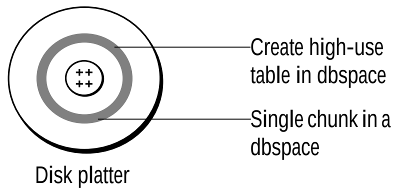


Figure 15-14
Disk Platter with High-Use Table Located on Middle Partitions

To place high-use tables on the middle partition of the disk, create a chunk using raw disk space that is composed of cylinders that reside midway between the spindle and the outer edge of the disk. Then create a dbspace with this same chunk as the initial and only chunk. When you create your high-use tables, use the IN clause of the CREATE TABLE statement to place them in the newly created dbspace.

For information about using raw disk space, see "Unbuffered or Buffered Disk Access on UNIX" on page 15-6.

Optimize Table-Extent Sizes

When two or more large, growing tables share a dbspace, their new extents can become interleaved. (See "Tblspaces" on page 15-34.) This interleaving creates gaps between the extents of any one table. (See Figure 15-11 on page 15-35.) Performance might suffer if disk seeks must span more than one extent. Work with the table owners to optimize the table extent sizes and thus limit head movement. For advice on how to alleviate this problem, see your *Performance Guide*. You can also consider placing the tables in separate dbspaces.

Sample Disk Layouts

When setting out to organize disk space, the database server administrator usually has one or more of the following objectives in mind:

- High performance
- High availability
- Ease and frequency of backup and restore

Meeting any one of these objectives has trade-offs. For example, configuring your system for high performance usually results in taking risks regarding the availability of data. The sections that follow present an example in which the database server administrator must make disk-layout choices given limited disk resources. These sections describe two different disk-layout solutions. The first solution represents a performance optimization, and the second solution represents an availability-and-restore optimization.

The setting for the sample disk layouts is a fictitious sporting-goods database that uses the structure (but not the volume) of the **stores_demo** database. In this example, the database server is configured to handle approximately 350 users and 3 gigabytes of data. The disk space resources are shown in the following table.

Disk Drive	Size of Drive	High Performance
Disk 1	1.5 gigabytes	No
Disk 2	2 gigabytes	Yes
Disk 3	2 gigabytes	Yes
Disk 4	1.5 gigabytes	No

The database includes two large tables: **cust_calls** and **items**. Assume that both of these tables contain more than 1,000,000 rows. The **cust_calls** table represents a record of all customer calls made to the distributor. The **items** table contains a line item of every order that the distributor ever shipped.

The database includes two high-use tables: **items** and **orders**. Both of these tables are subject to constant access from users around the country.

The remaining tables are low-volume tables that the database server uses to look up data such as postal code or manufacturer.

Table Name	Maximum Size	Access Rate
cust_calls	1.5 gigabytes	Low
items	0.5 gigabytes	High
orders	50 megabytes	High
customers	50 megabytes	Low
stock	50 megabytes	Low
catalog	50 megabytes	Low
manufact	50 megabytes	Low
state	50 megabytes	Low
call_type	50 megabytes	Low

Sample Layout When Performance Is Highest Priority

Figure 15-15 shows a disk layout optimized for performance. This disk layout uses the following strategies to improve performance:

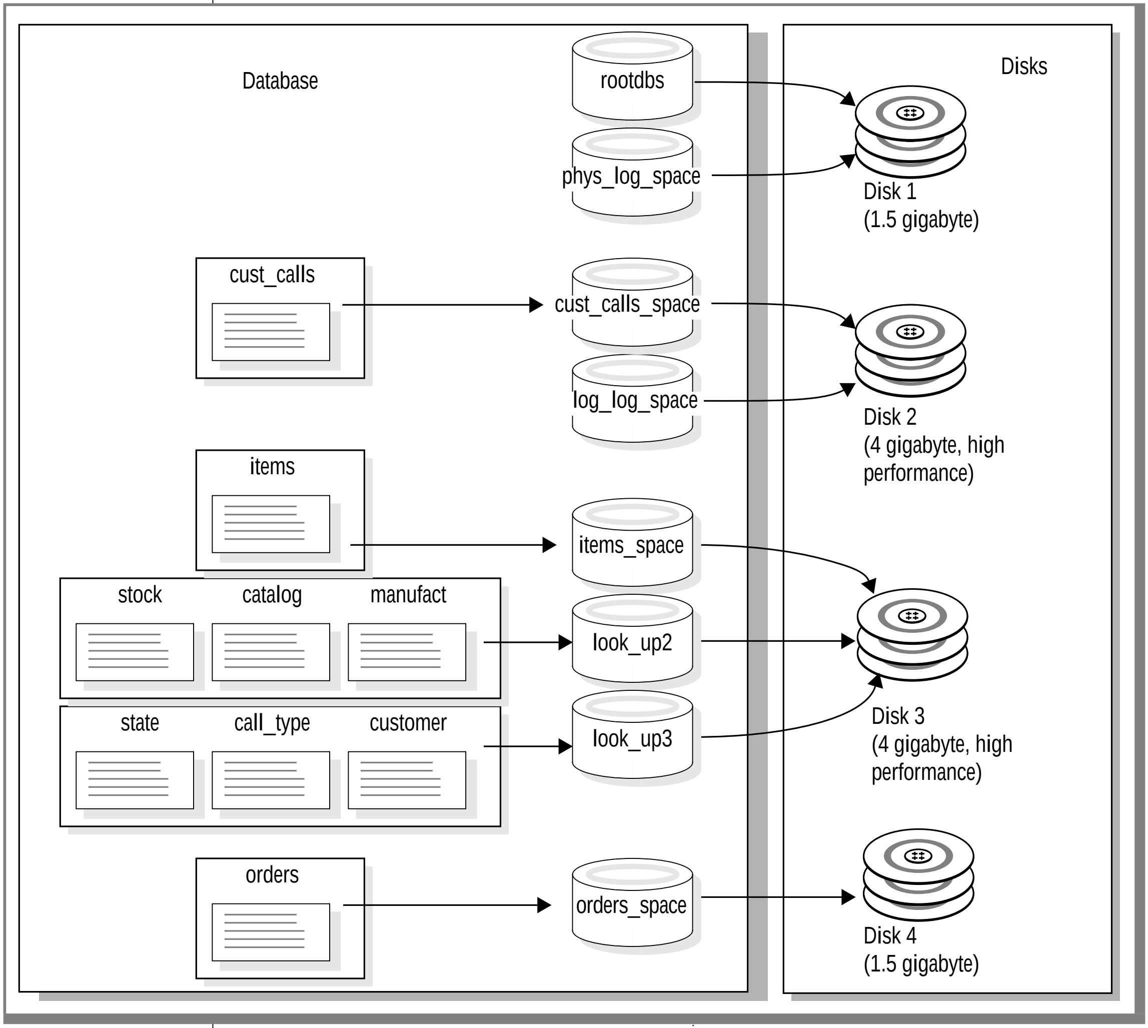
- Migration of the logical log from the rootdbs dbspace to a dbspace on a separate disk

  This strategy separates the logical log and the physical log and reduces contention for the root dbspace.

- Location of the two tables that undergo the highest use in dbspaces on separate disks

  Neither of these disks stores the logical log or the physical log. Ideally you could store each of the **items** and **orders** tables on a separate high-performance disk. However, in the present scenario, this strategy is not possible because one of the high-performance disks is needed to store the very large **cust_calls** table (the other two disks are too small for this task).

Figure 15-15
Disk Layout Optimized for Performance

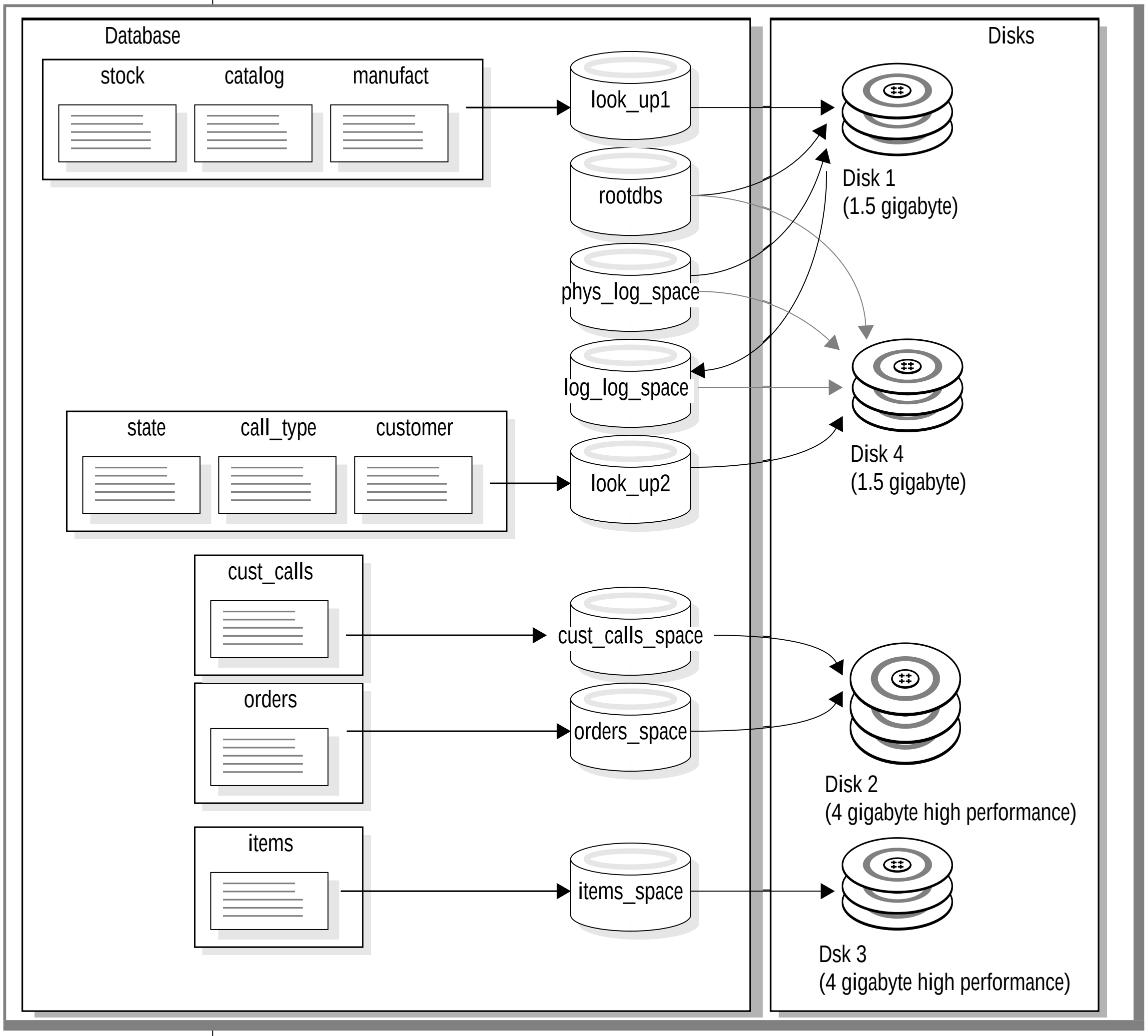


Sample Layout When Availability Is Highest Priority

The weakness of the previous disk layout is that if either disk 1 or disk 2 fails, the whole database server goes down until you restore the dbspaces on these disks from backups. In other words, the disk layout is poor with respect to availability.

An alternative disk layout that optimizes for availability is shown in Figure 15-16. This layout mirrors all the critical data spaces (the system catalog tables, the physical log, and the logical log) to a separate disk. Ideally you could separate the logical log and physical log (as in the previous layout) and mirror each disk to its own mirror disk. However, in this scenario the required number of disks does not exist; therefore, the logical log and the physical log both reside in the root dbspace.

Figure 15-16
Disk Layout Optimized for Availability



Logical-Volume Manager

A logical-volume manager (LVM) is a utility that allows you to manage your disk space through logical volumes.

Many computer manufacturers ship their computers with a proprietary LVM. You can use the database server to store and retrieve data on disks that are managed by most proprietary LVMs. Logical-volume managers provide some advantages and some disadvantages, as discussed in the remainder of this section.

Most LVMs can manage multiple gigabytes of disk space. On Extended Parallel Server, the maximum chunk size can be 4 gigabytes or even larger for 64-bit platforms. To determine which chunk size your platform supports, refer to your machine notes file.

Because LVMs allow you to partition a disk drive into multiple volumes, you can control where data is placed on a given disk. You can improve performance by defining a volume that consists of the middle-most cylinders of a disk drive and placing high-use tables in that volume. For more information, see "Place High-Use Tables on Middle Partition of Disk" on page 15-46. (Technically, you do not place a table directly in a volume. You must first allocate a chunk as a volume, then assign the chunk to a dbspace, and finally place the table in the dbspace. For more information, see "Control of Where Data Is Stored" on page 15-15.)

You can also improve performance by using a logical volume manager to define a volume that spreads across multiple disks and then placing a table in that volume. This strategy helps reduce contention between programs that access the same table, as explained in "Place High-Use Tables on Middle Partition of Disk" on page 15-46.

Many logical volume managers also allow a degree of flexibility that standard operating-system format utilities do not. One such feature is the ability to reposition logical volumes after you define them. Thus getting the layout of your disk space right the first time is not so critical as with operating-system format utilities.

LVMs often provide operating-system-level mirroring facilities. For more information, see "Alternatives to Mirroring" on page 25-6.

Figure 15-17 illustrates the role of fragments in specifying the location of data.

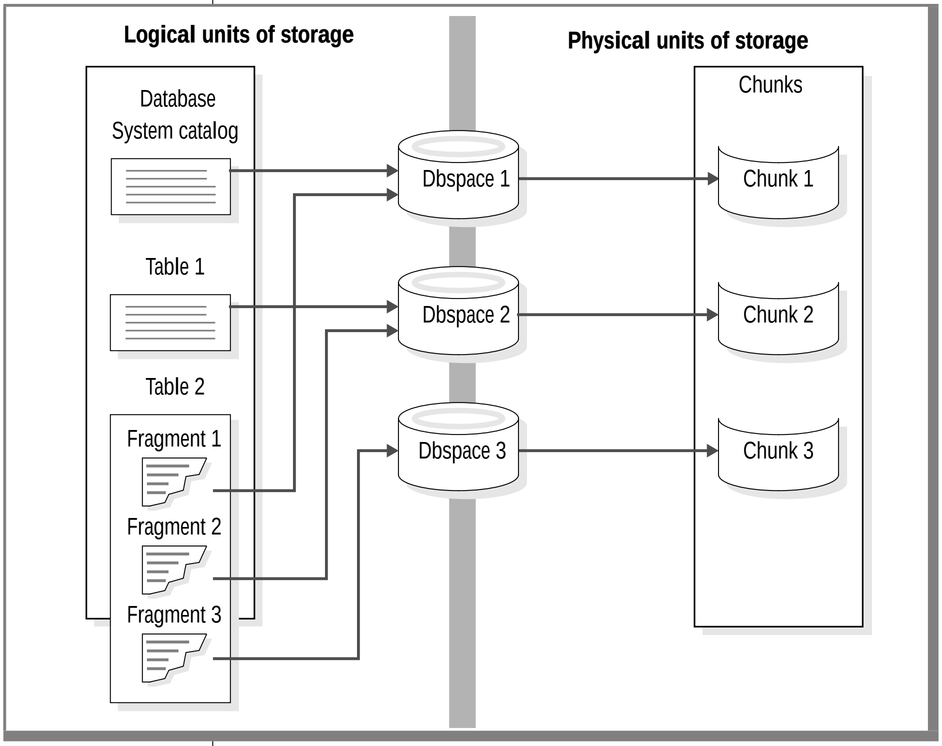


Figure 15-17
Dbspaces That Link Logical Units (Including Table Fragments) and Physical Units of Storage

Usually you fragment a table when you initially create it. The CREATE TABLE statement takes one of the following forms:

```
CREATE TABLE tablename ... FRAGMENT BY ROUND ROBIN IN
dbspace1, dbspace2, dbspace3;


CREATE TABLE tablename ...FRAGMENT BY EXPRESSION
    <Expression 1> in dbspace1,
    <Expression 2> in dbspace2,
    <Expression 3> in dbspace3;
```

The FRAGMENT BY ROUND ROBIN and FRAGMENT BY EXPRESSION keywords refer to two different distribution schemes. Both statements associate fragments with dbspaces. For more information on fragmentation schemes, refer to the *Informix Guide to Database Design and Implementation*

Chapter

Managing Disk Space

16

In This Chapter

This chapter provides the instructions that you need to manage effectively the disk spaces and data that the database server controls. It assumes you are familiar with the terms and concepts contained in Chapter 15, “Data Storage.”

This chapter covers the following topics:

- Initializing disk space
- Allocating disk space
- Setting configuration variables related to disk management
- Backing up after you change the physical schema
- Managing chunks and storage spaces
  - Creating and dropping dbspaces and dbslices
  - Allocating, adding, and dropping chunks from dbspaces
  - Altering dbslices
  - Optimizing blobpage size
- Skipping inaccessible fragments
- Monitoring disk space
- Monitoring simple-large-object data

Your *Performance Guide* also contains information about managing disk space. In particular, it describes how to eliminate interleaved extents and how to reclaim space in an empty extent.

Initializing Disk Space

Disk-space initialization uses the values stored in the configuration file to create the initial chunk of the root dbspace on disk and to initialize shared memory. When you initialize disk space, shared memory is automatically initialized for you as part of the process.

Typically, you initialize disk space just once in the life of an database server. This action occurs when you bring the database server on-line for the first time.

Warning: *When you initialize the database server disk space, you overwrite whatever is on that disk space. If you reinitialize disk space for an existing database server, all data in the earlier database server instance becomes inaccessible and, in effect, is destroyed.*

For information on initializing the database server, see "Initializing Disk Space" on page 9-5.

Allocating Disk Space

This section explains how to allocate disk space for the database server. Read the following sections before you allocate disk space:

- "Unbuffered or Buffered Disk Access on UNIX" on page 15-6
- "Amount of Disk Space Needed to Store Data" on page 15-36
- "Disk-Layout Guidelines" on page 15-39

Before the database server can use disk space, you might need to perform these tasks:

- Initializing disk space
- Creating a dbspace
- Adding a chunk to an existing dbspace
- Mirroring an existing dbspace

You can allocate either an empty file or a portion of raw disk for database server disk space.

If you allocate raw disk space, Informix recommends that you use the **ln** command to create a link between the character-special device name and another filename. For more information on this topic, see "Creating Standard Device Names" on page 16-12.

Using a UNIX file and its inherent operating-system interface for database server disk space also is referred to as using *cooked space*.

Specifying an Offset

When you allocate a chunk of disk space to the database server, you might want to specify an offset for one of the following two purposes:

- To prevent the database server from overwriting the partition information
- To define multiple chunks on a partition, disk device, or cooked file

Many computer systems and some disk-drive manufacturers keep information for a physical disk drive on the drive itself. This information is sometimes referred to as a volume table of contents (VTOC) or disk label. (For convenience, it is referred to here as the VTOC.) The VTOC is commonly stored on the first track of the drive. A table of alternate sectors and bad-sector mappings (also called revectoring table) might also be stored on the first track.

If you plan to allocate partitions at the start of a disk, you might need to use offsets to prevent the database server from overwriting critical information required by the operating system. For the exact offset required, refer to your disk-drive manuals.

Warning: *If you are running two or more instances of the database server, be extremely careful not to define chunks that overlap. Overlapping chunks can cause the database server to overwrite data in one chunk with unrelated data from an overlapping chunk. This overwrite effectively destroys overlapping data.*

Specifying an Offset for the Initial Chunk of Root Dbspace

For the initial chunk of root dbspace and its mirror, if it has one, specify the offsets with the ROOTOFFSET and MIRROROFFSET parameters, respectively. For more information, see the chapter on configuration parameters in the *Administrator's Reference*.

Specifying an Offset for Additional Chunks

To specify an offset for additional chunks of database server space, you must supply the offset as a parameter when you assign the space to the database server with the **onutil** utility.

For more information on specifying an offset for chunks of database server space, see "Creating a Dbspace" on page 16-13.

Using Offsets to Create Multiple Chunks

You can create multiple chunks from a disk partition, disk device, or file, by specifying offsets and assigning chunks that are smaller than the total space available. The offset specifies the beginning location of a chunk. The database server determines the location of the last byte of the chunk by adding the size of the chunk to the offset.

For the first chunk, assign any initial offset, if necessary, and specify the size as an amount that is less than the total size of the allocated disk space. For each additional chunk specify the offset to include the sizes of all previously assigned chunks, plus the initial offset, and assign a size that is less than or equal to the amount of space remaining in the allocation.

Allocating a File for Disk Space on UNIX

To allocate a file for database server disk space on UNIX, log in as user **informix** and concatenate null to the filename that the database server will use for disk space. The file should have permissions set to `660` (rw-rw----). Group and owner must be set to **informix**. Figure 16-1 illustrates these steps and allocates the file **/usr/data/my_chunk** for disk space.

Figure 16-1
Preparing Cooked File Space

Step	Command	Comments
1.	`su informix`	Log in as user **informix**. (Enter the password.)
2.	`cd /usr/data`	Change directories to the directory where the cooked space will reside.
3.	`cat /dev/null > my_chunk`	Create your chunk by concatenating null to a file (in this example, a file named **my_chunk**).
4.	`chmod 660 my_chunk`	Set the permissions of the file to `660` (rw-rw----).
5.	`ls -lg my_chunk -rw-rw---- 1 informix informix 0 Oct 12 13:43 my_chunk`	Use `ls -l` if you are using System V UNIX. Verify that both group and owner of the file are **informix**. You should see something like this line (which has wrapped around).

For information on how to create a dbspace using the file you have allocated, refer to "Creating a Dbspace" on page 16-13.

Once you have allocated the file space, you can create the dbspace or other storage space as you normally would, using **onutil.** For information on how to create a dbspace or a dbslice, refer to "Creating a Dbspace with onutil" on page 16-16 and "Creating Dbslices" on page 16-16.

You must also follow the preceding steps prior to adding a chunk to a dbspace.

Allocating Raw Disk Space on UNIX

For specific instructions on how to allocate raw disk space on UNIX, see your operating-system documentation.

In general, to create raw disk space, you can either repartition your disks or unmount an existing file system. In either case, take proper precautions to back up any files before you unmount the device. (See "Unbuffered or Buffered Disk Access on UNIX" on page 15-6.)

Change the group and owner of the character-special devices to **informix**. The filename of the character-special device usually begins with the letter *r*.

Verify that the operating-system permissions on the character-special devices are `crw-rw----`.

Warning: *After you create the raw device that the database server uses for disk space, carefully heed the following warnings:*

- *Do not create file systems on the same raw device that you allocate for the database server disk space.*
- *Do not use the same raw device as swap space that you allocate for the database server disk space.*

Create a link between the character-special device name and another filename with the UNIX link command, usually **ln**.

The link enables you to replace quickly the disk where the chunk is located. The convenience becomes important if you need to restore your database server data. The restore process requires all chunks that were accessible at the time of the last dbspace backup to be accessible when you perform the restore. The link means that you can replace a failed device with another device and link the new device pathname to the same filename that you previously created for the failed device. You do not need to wait for the original device to be repaired.

Execute the command **ls -lg** (**ls -l** on System V UNIX) on your device directory to verify that both the devices and the links exist. The following example shows links to raw devices. If your operating system does not support symbolic links, hard links will work as well.

```
% ls -lg
crw-rw---  /dev/rxy0h
crw-rw---  /dev/rxy0a
lrwxrwxrwx /dev/my_root@->/dev/rxy0h
lrwxrwxrwx /dev/raw_dev2@->/dev/rxy0a
```

Configuring Disk Space for Multiple Coservers

Configuring your disks is possibly the most important task for obtaining optimum performance with VLDBs. Disk I/O is the longest portion of the response time for an SQL operation that scans a large amount of data. Extended Parallel Server offers the advantage of parallel access to multiple disks spread across many coservers.

Your goal should be to make it easy for a DBA to administer a large database server and to ensure that tables can be fragmented appropriately across disks and coservers for fully parallel processing. To accommodate multiple coservers and multiple nodes, perform the following steps:

- Create the dbspaces that the coservers manage on as many physical disk drives as possible. This step maximizes parallel I/O access to multiple disks. To prevent I/O access problem, if possible make sure that each disk can be accessed by only one coserver.
- Create standard device names and chunk path names across all coservers. Although you are not required to create standard device names across all coservers, standard device names make managing the database server easier. You must use unique chunk path names for each disk on a node. Be careful not to duplicate chunk path names.

Creating Standard Device Names

Use symbolic links to assign abbreviated standard device names. To create a link between the character-special device name and another filename, use the link command (usually **ln)**.

To verify that both the devices and the links exist, execute the command **ls -l** (**ls -lg** on BSD) on your device directory. The following example shows links to raw devices. If your operating system does not support symbolic links, hard links work as well.

```
% ls -lg
crw-rw---  /dev/rxy0h
crw-rw---  /dev/rxy0a
lrwxrwxrwx /dev/my_root@->/dev/rxy0h
lrwxrwxrwx /dev/raw_dev2@->/dev/rxy0a
```

Extended Parallel Server requires standard device names across all coservers.

Setting Up Disk Access Across Nodes

The file system on which the **INFORMIXDIR** directory is installed should be exported to, and mounted by, all nodes that are defined for the database server. In addition, you must replicate the following utilities on each node:

- **oninit**
- **onmode**
- **onstat**

Place the directory that contains these copied utilities before **$INFORMIXDIR/etc** in the search path because the **INFORMIXDIR** directory might be on another node.

Backing Up After You Change the Physical Schema

You must perform a level-0 backup of the root dbspace and the modified storage spaces to ensure that you can restore the data when you:

- add or drop mirroring.
- add, move, drop, or resize a logical-log file.
- change the size or location of the physical log.
- change your storage-manager configuration.
- add or drop a dbspace, dbslice, or logslice.
- add, move, or drop a chunk to a dbspace.

You must perform a level-0 backup of the modified storage spaces to ensure that you can restore the data when you convert a raw, static, or operational table to standard. This backup ensures that the unlogged data is restorable before you switch to a logging table type.

Creating a Dbspace

This section explains how to create a standard dbspace and a temporary dbspace. See "Dbspaces" on page 15-14 and "Temporary Dbspaces" on page 15-18.

You can use **onutil** to create a dbspace.

Before you create a dbspace, you must first allocate disk space as described in "Allocating Disk Space" on page 16-6.

You must be logged in as user **informix** or **root** to create a dbspace.

If you are creating a standard dbspace, the database server can be in on-line mode. The newly added dbspace (and its mirror, if one exists) is available immediately.

If you are using mirroring, you can mirror the dbspace when you create it. Mirroring takes effect immediately.

When the initial chunk of the dbspace that you are creating is a cooked file, the database server verifies that the disk space is sufficient for the initial chunk. If the size of the chunk is greater than the available space on the disk, a message is displayed, and no dbspace is created. However, the cooked file that the database server created for the initial chunk is not removed. Its size represents the space left on your file system before you created the dbspace. Remove this file to reclaim the space.

Specifying Pathnames for Dbspaces

Specify an explicit pathname for the initial chunk of the dbspace as follows:

- If you are using raw disks, Informix recommends that you use a linked pathname. (See "Creating Standard Device Names" on page 16-12.)
- If you are using a file for database server disk space, the pathname is the complete path and filename.

Specifying Names and Maximum Number of Storage Spaces

Specify a dbspace name of up to 18 characters. The name must be unique and begin with a letter or underscore. You can use letters, digits, and underscores in the name.

If you use a CONFIGSIZE value of `LARGE`, you can create up to 8192 dbspaces. Use the MAX_DBSPACES parameter to increase the maximum number of dbspaces on the system to 32,767. For more information on MAX_DBSPACES, see the chapter on configuration parameters in the *Administrator's Reference* and "Increasing the Maximum Number of Dbspaces, Chunks, or Dbslices" on page 16-19.

Backing Up the New Dbspace

After you create the dbspace, you must perform a level-0 backup of the root dbspace and the new dbspace.

Creating a Temporary Dbspace

To specify where to allocate the temporary files, create temporary dbspaces.

To define temporary dbspaces

1. Use **onutil** CREATE TEMP DBSPACE.

   For more information, refer to "Creating a Dbspace with onutil" on page 16-16.
2. Use the **DBSPACETEMP** environment variables or the DBSPACETEMP configuration parameter to specify the dbspaces that the database server can use for temporary storage.

   For further information on DBSPACETEMP, refer to the chapter on configuration parameters in the *Administrator's Reference*.
3. If you create more than one temporary dbspace, the dbspaces should reside on separate disks to optimize the I/O.

If you are creating a temporary dbspace, you must make the database server aware of the existence of the newly created temporary dbspace by setting the DBSPACETEMP configuration variable, the **DBSPACETEMP** environment variable, or both. The database server does not begin to use the temporary dbspace until you take both of the following steps:

- Set the DBSPACETEMP configuration parameter, the **DBSPACETEMP** environment variable, or both.
- Reinitialize the database server.

Creating a Dbspace with onutil

You can create a new dbspace or temporary dbspace on a specific coserver.

When you create separate dbspaces without using a dbslice, a specific coserver owns and manages each dbspace. The physical disk on which the dbspace resides belongs to the node on which the coserver executes. This coserver is referred to as the *home coserver.*

The following example shows the **onutil** CREATE DBSPACE command to create a dbspace that coserver **eds.2** owns:

```
% onutil
1> CREATE DBSPACE acctg_dbsp
2> CHUNK '/work/dbspaces/dbs_0'
3> OFFSET 0 size 1500
4> COSERVER eds.2;
```

The above example assumes that one coserver exists on each node. For more information on **onutil**, refer to the utilities chapter in the *Administrator's Reference*.

Creating Dbslices

You can create dbslices with the database server in on-line or quiescent mode.

To obtain the maximum performance benefit from multiple coservers, Informix recommends that you fragment all tables, except very small tables, across all available coservers. Dbslices allow you to manage a set of dbspaces in parallel across multiple coservers. For instance, rather than issuing a separate **onutil** CREATE DBSPACE command for each dbspace that you intend to use for a fragmented table, you can use the **onutil** CREATE DBSLICE command to create them all. The dbspaces that you create with **onutil** follow the naming convention that you specify as an argument to the CREATE DBSLICE command, as the following example indicates:

```
% onutil
1> CREATE DBSLICE dbsl
2>  FROM COGROUP cogroup_all
3>  CHUNK '/dev/dbsl_all'
4>  OFFSET 1024 SIZE 1024;
```

The above example assumes that one coserver exists on each node. If the **cogroup_all** coserver group contains coserver **eds.1** through coserver **eds.16**, this command creates the following dbspaces on the indicated chunks.

Coserver	Dbspace_identifier	Primary Chunk	Offset
eds.1	dbsl.1	/dev/dbsl_all	1024
eds.2	dbsl.2	/dev/dbsl_all	1024
eds.3	dbsl.3	/dev/dbsl_all	1024
...			
eds.16	dbsl.16	/dev/dbsl_all	1024

Naming Dbslices

Specify a dbslice name of up to 18 characters. The name must begin with a letter or underscore. You can use letters, digits, and underscores in the name.

For more information on dbslices and their relationship to dbspaces, see Chapter 1, "Introducing the Database Server." For more information on the allocation and management of dbslices and dbspaces, see **onutil** in the utilities chapter of the *Administrator's Reference*.

Increasing the Number of Dbslices

If you use a CONFIGSIZE value of LARGE, you can create up to 512 dbslices. Use the MAX_DBSLICES parameter to increase the maximum number of dbslices on the system to 2047. For more information on MAX_DBSLICES, see the chapter on configuration parameters in the *Administrator's Reference* and "Increasing the Maximum Number of Dbspaces, Chunks, or Dbslices" on page 16-19.

Backing Up the New Dbslice

After you create or alter a dbslice, you must perform a complete level-0 backup on all coservers.

Altering a Dbslice

To alter a dbslice, add dbspaces to it. You can either add dbspaces to a dbslice on the same coservers on which they were created or expand a dbslice to another coserver. These new dbspaces are not automatically visible to the existing tables in the dbslice. Use the SQL statement ALTER FRAGMENT to refragment the tables to enable them to use the new dbspaces in the dbslice. After you alter the dbslice, you can store new tables in either the new or old dbspaces.

Suppose you add coservers **eds.4** and **eds.5** to the dbslice **dbsl** that you created in the previous example. (See "Creating Dbslices" on page 16-16.) To expand the dbslice to the new coservers, use the following command:

```
% onutil
ALTER DBSLICE dbsl ADD DBSPACE
FROM COGROUP eds.%r(4..5)
CHUNK '/dev/dbsl_45' SIZE 2048;
```

This command creates two 2048-kilobyte dbspaces in the dbslice, one each on coservers **eds.4** and **eds.5**. If this dbslice also contains a logslice, use the **onutil** ALTER LOGSLICE command to add logical logs to the logslice. The above example assumes that one coserver exists on each node. For more information on **onutil**, see the utilities chapter in the *Administrator's Reference*.

Increasing the Maximum Number of Dbspaces, Chunks, or Dbslices

In Version 8.2, the CONFIGSIZE configuration parameter determined the maximum number of dbspaces, chunks, and dbslices. In Version 8.3, use the MAX_DBSPACES, MAX_CHUNKS, and MAX_DBSLICES configuration parameters to increase the maximum number of dbspaces, chunks, or dbslices that the database server can maintain. You can increase the maximum number of dbspaces, chunks, or dbslices from the number that CONFIGSIZE specified up to the new maximums listed below.

Storage Space	Old Maximum (with CONFIGSIZE=LARGE)	Current Maximum Number
dbslices	512	2047
dbspaces	8192	32767
chunks	8192	32767

Like CONFIGSIZE, these MAX* parameters take effect at shared memory initialization.

The database server ignores these MAX* parameters if their values are smaller than the values that are set by CONFIGSIZE. The database server also ignores their values if they are smaller than the maximum previously set by CONFIGSIZE. Once you successfully increase the maximum number of dbspaces, chunks, or dbslices, you cannot reduce it (without reinitializing the database server). If the value of the parameter is greater than the new maximum number listed above, the database server uses the new maximum number instead.

Converting from Version 8.2 to Version 8.3

If you are converting from Version 8.2 to Version 8.3, Informix recommends that you complete the conversion first. Set these MAX* parameters in the ONCONFIG file and then bring up the Version 8.3 database server. The database server expands some structures (such as the safewrite area) in the root dbspace to accommodate the new maximum values. If the database server expands the structures successfully, the following messages appear in the on-line message log:

```
# If MAX_DBSLICES has been set to 1024:
Configuration has been grown to handle up to 1024 dbslices.

# If MAX_CHUNKS has been set to 32767:
Configuration has been grown to handle up to 32767 chunks.
```

Recovering from Errors

If the database server fails to expand the structures, the following message appears in the on-line message log, and the database server halts:

```
error:  Insufficient available disk in the root dbspace to
increase the entire Configuration save area.
```

If this error occurs, you must reset CONFIGSIZE, MAX_CHUNKS, MAX_DBSPACES, or MAX_DBSLICES to a lower value and restart the database server. For more information, see the chapter on the configuration parameters in the *Administrator's Reference*.

Adding a Chunk to a Dbspace

If one of your dbspaces is becoming full, you might want to add a new chunk. Before you do, however, you must first allocate disk space as described in "Allocating Disk Space" on page 16-6.

You add a *chunk* when you need to increase the amount of disk space allocated to a storage space. To add a chunk, you must be logged in as user **informix** or **root**.

If you are adding a chunk to a mirrored storage space, you must also add a mirrored chunk.

When you add a chunk that is allocated as a cooked file, the database server verifies that the disk space is sufficient for the new chunk by creating and then removing a file of the size requested. If the size of the chunk is greater than the available space on the disk, the database server might inadvertently fill your file system in the process of verifying available disk space.

To add a chunk to a dbspace, use the **onutil** utility.

Important: *You can add a chunk while the database server is in on-line or quiescent mode. The newly added chunk (and its associated mirror, if one exists) is available immediately.*

Backing Up the New Chunk

After you create the new chunk, you must perform a level-0 backup of the root dbspace and the dbspace that contains the chunk.

Naming Chunks and Storage Spaces

You must specify an explicit pathname for the chunk. For more information, see "Creating a Dbspace" on page 16-13.

The name is case insensitive unless you use quotes around it. By default, the database server converts uppercase characters in the name to lowercase. If you want to use uppercase in names, put quotes around them and set the **DELIMIDENT** environment variable to `ON`.

The **onutil** utility does not support the **DELIMIDENT** environment variable. For the naming rules, see "Specifying Names and Maximum Number of Storage Spaces" on page 16-14.

Limiting Chunk Size and Number

The maximum number of chunks that you can allocate for a given Extended Parallel Server depends on the value of the CONFIGSIZE or MAX_CHUNKS configuration parameter. If CONFIGSIZE is set to `LARGE` and MAX_CHUNKS is not set, the maximum number of chunks is 8192. However, you can use the MAX_CHUNKS parameter to specify up to 32,767 chunks. For more information on the CONFIGSIZE and MAX_CHUNKS configuration parameters, see the *Administrator's Reference* and "Increasing the Maximum Number of Dbspaces, Chunks, or Dbslices" on page 16-19.

This maximum number of chunks is the total number of chunks across all coservers. For example, if you configure 10 coservers, the maximum number of chunks is 819 per coserver (if CONFIGSIZE is set to `LARGE`) or 3276 per coserver (if MAX_CHUNKS is set to `32,767`).

Adding a Chunk with onutil

To add a chunk to a dbspace, use the **onutil** ALTER DBSPACE command.

This example adds a chunk of 5000 kilobytes of raw disk space, at an offset of 5200 kilobytes, to dbspace **dbspc3** on coserver 2.

```
% onutil
1> ALTER DBSPACE dbsp3
2> CHUNK '/dev/dbsl_45' OFFSET 5200 size 5000
3> COSERVER eds.2;
```

Loading Data Into a Table

You can load data into an existing table in the following ways.

Method to Load Data	TEXT or BYTE Data	Reference
DB-Access LOAD statement	Yes	LOAD statement in the *Informix Guide to SQL: Syntax*
dbload utility	Yes	*Informix Migration Guide*
dbimport utility	No	*Informix Migration Guide*
Informix ESQL/C programs	Yes	*Informix ESQL/C Programmer's Manual*
From external tables	Yes	Chapter on loading with external tables in the *Administrator's Reference*

Dropping a Chunk

Use **onutil** ALTER DBSPACE to drop a chunk from a dbspace.

Before you drop a chunk, ensure that the database server is in the correct mode, using the following table as a guideline.

Chunk Type	Database Server in On-line Mode	Database Server in Quiescent Mode	Database Server in Off-line Mode
Dbspace chunk	Yes	Yes	No
Temporary dbspace chunk	Yes	Yes	No

Verifying Whether a Chunk Is Empty

To drop a chunk successfully from a dbspace with either of these utilities, the chunk must not contain any data. All pages other than overhead pages must be freed. If any pages remain allocated to nonoverhead entities, the utility returns the following error:

```
Chunk is not empty.
```

If this situation occurs, you must determine which table or other entity still occupies space in the chunk by executing **onutil** CHECK SPACE. Usually, the pages can be removed when you drop the table that owns them. Then reenter the utility command.

Dropping a Chunk from a Dbspace with onutil

Use the **onutil** ALTER DBSPACE command with the DROP CHUNK clause to drop a chunk. If you drop a chunk that is mirrored, the mirrored chunk is also dropped.

The following example drops a chunk from **dbsp3**:

```
% onutil
1> ALTER DBSPACE dbspc3
2> DROP CHUNK '/dev/raw_dev1'
3> OFFSET 300;
```

You cannot drop the initial chunk of a dbspace with the syntax in the previous example. Instead, you must drop the dbspace. For more information, refer to "Dropping a Dbspace with onutil" on page 16-26.

Use the `chunk` column of **xctl onstat -d** to determine which chunk is the initial chunk of a dbspace. For more information on **onstat**, see the utilities chapter of the *Administrator's Reference*.

After you drop the chunk, you must perform a level-0 backup of the root dbspace and the modified dbspace on the affected coserver.

Dropping a Storage Space

Use **onutil** DROP DBSPACE to drop a storage space.

You must be logged in as **root** or **informix** to drop a storage space.

Before you drop a storage space, ensure that the database server is in either on-line or quiescent mode. You cannot drop a storage space when it is in off-line mode.

Preparing to Drop a Storage Space

Before you drop a dbspace, you must first drop all databases and tables that you previously created in that dbspace.

Execute **onutil** CHECK SPACE. You cannot drop the root dbspace or dbslice.

Backing Up After Dropping a Storage Space

If you create a storage space with the same name as the deleted storage space, perform another level-0 backup to ensure that future restores do not confuse the new storage space with the old one.

If you are using ON-Bar for your backup and restore system, see the *Backup and Restore Guide*. If you are using ON-Archive as your backup and restore tool, refer to your *Archive and Backup Guide*.

Warning: *After you drop a dbspace or blobspace, the newly freed chunks are available for reassignment to other dbspaces or blobspaces. However, before you reassign the newly freed chunks, you must perform a level-0 backup of the root dbspace and the modified dbspace or blobspace. If you do not perform this backup, and you subsequently need to perform a restore, the restore might fail because the dbspace-backup reserved pages are not up-to-date.*

Dropping a Mirrored Storage Space

If you drop a storage space that is mirrored, the mirror spaces are also dropped.

If you want to drop only a storage-space mirror, turn off mirroring. (See "Ending Mirroring with onutil" on page 26-8.) This action drops the dbspace mirrors and frees the chunks for other uses.

Dropping a Dbspace with onutil

To drop a storage space with **onutil**, use the DROP DBSPACE option, as the following examples illustrate.

This example drops a dbspace called **dbspce5** and its mirrors.

```
% onutil
1> DROP DBSPACE dbspc5;
```

This example drops a dbspace that is part of a dbslice called **dbsl**.

```
% onutil
1> DROP DBSPACE dbsl.16;
```

After you drop the dbspace, you must perform a level-0 backup of the root dbspace and the modified dbspace on the affected coserver.

Dropping Dbslices

The database server must be in on-line or quiescent mode before you can drop a dbslice.

Dbslices allow you to manage a set of dbspaces in parallel across multiple coservers. For instance, rather than issuing a separate **onutil** DROP DBSPACE command for each dbspace that you intend to drop after you have removed a fragmented table, you can use the **onutil** DROP DBSLICE command to drop them all. The dbspaces that you drop with **onutil** must follow the naming convention that you specified when you created the dbslice.

Warning: *After you drop a dbslice, perform a level-0 backup on the coservers that contained the dbspaces that made up the dbslice.Then the newly freed chunks are available for reassignment to other dbspaces. Otherwise, a restore operation on reassigned chunks might fail.*

If you drop a dbslice that is mirrored, the mirror dbslice is also dropped.

You must be logged in as **root** or **informix** to drop a dbslice. On Windows NT, you must be a member of the **Informix-Admin** group.

For more information on dbslices and their relationship to dbspaces, see "Dbslices" on page 15-19. For more information on the allocation and management of dbslices and dbspaces, see the information on **onutil** in the *Administrator's Reference*.

Skipping Inaccessible Fragments

One benefit that fragmentation provides is the ability to skip table fragments that are unavailable during an I/O operation. For example, a query can proceed even when a fragment is located on a chunk that is currently down as a result of a disk failure. When this situation occurs, a disk failure affects only a portion of the data in the fragmented table. By contrast, tables that are not fragmented can become completely inaccessible if they are located on a disk that fails.

This functionality is controlled as follows:

- By the database server administrator with the DATASKIP configuration parameter
- By individual applications with the SET DATASKIP statement

Using the DATASKIP Configuration Parameter

You can set the DATASKIP parameter to OFF, ALL, or ON *dbspace_list*. OFF means that the database server does not skip any fragments. If a fragment is unavailable, the query returns an error. ALL indicates that any unavailable fragment is skipped. ON *dbspace_list* instructs the database server to skip any fragments that are located in the specified dbspaces.

Using the Dataskip Feature of onutil

Use the dataskip feature of the **onutil** utility to specify the dbspaces that are to be skipped when they are unavailable. The following example sets the DATASKIP parameter so that the database server skips two dbspaces in the dbslice **acctg_dbslc**:

```
% onutil
1> SET DATASKIP ON acctg_dbslc.1,
2> acctg_dbslc.2;
```

Using onstat to Check Dataskip Status

Use the **onstat** utility to list the dbspaces currently affected by the dataskip feature. The **-f** option lists the dbspaces that were set with the DATASKIP configuration parameter. When you execute **onstat -f**, you see one of the following messages:

```
dataskip is OFF for all dbspaces

dataskip is ON for all dbspaces

databskip is ON for dbspaces:
    dbspace1 dbspace2 ...
```

Using the SQL Statement SET DATASKIP

An application can use the SQL statement SET DATASKIP TO control whether a fragment should be skipped if it is unavailable. Applications should include this statement only in limited circumstances because it causes queries to return different results, depending on the availability of the underlying fragments. Like the configuration parameter DATASKIP, the SET DATASKIP statement accepts a list of dbspaces that indicate to the database server which fragments to skip. For example, suppose that an application programmer included the following statement at the beginning of an application:

```
SET DATASKIP ON dbspace1, dbspace5
```

This statement causes the database server to skip **dbspace1** or **dbspace5** whenever both of these conditions are met:

- The application attempts to access one of the dbspaces.
- The database server finds that one of the dbspaces is unavailable.

If the database server finds that both **dbspace1** and **dbspace5** are unavailable, it skips both dbspaces.

The DEFAULT setting for the SET DATASKIP statement allows a database server administrator to control the dataskip feature. Suppose that an application developer includes the following statement in an application:

```
SET DATASKIP DEFAULT
```

When a query is executed subsequent to this SQL statement, the database server checks the value of the configuration parameter DATASKIP. Encouraging end users to use this setting allows the database server administrator to specify which dbspaces are to be skipped as soon as the database server administrator becomes aware that one or more dbspaces are unavailable.

Effect of the Dataskip Feature on Transactions

If you turn the dataskip feature on, a SELECT statement always executes. In addition, an INSERT statement always succeeds if the table is fragmented by round-robin and at least one fragment is on-line. However, the database server *does not* complete operations that write to the database if a possibility exists that such operations might compromise the integrity of the database. The following operations fail:

- All UPDATE and DELETE operations where the database server cannot eliminate the down fragments

  If the database server *can* eliminate the down fragments, the update or delete is successful, but this outcome is independent of the DATASKIP setting.
- An INSERT operation for a table fragmented according to an expression-based distribution scheme where the appropriate fragment is down
- Any operation that involves referential constraint checking if the constraint involves data in a down fragment

  For example, if an application deletes a row that has child rows, the child rows must also be available for deletion.
- Any operation that affects an index value (for example, updates to a column that is indexed) where the index in question is located in a down chunk

Determining When to Use Dataskip

Use this feature sparingly and with caution because the results are always suspect. Consider using it in the following situations:

- You can accept the compromised integrity of transactions.
- You can determine that the integrity of the transaction is not compromised.

The latter task can be difficult and time consuming.

Determining When to Skip Selected Fragments

In certain circumstances, you might want the database server to skip some fragments, but not others. This usually occurs in the following situations:

- Fragments can be skipped because they do not contribute significantly to a query result.
- Certain fragments are down, and you decide that skipping these fragments and returning a limited amount of data is preferable to canceling a query.

When you want to skip fragments, use the ON *dbspace-list* setting to specify a list of dbspaces with the fragments that the database server should skip.

Determining When to Skip All Fragments

Setting the DATASKIP configuration parameter to ALL causes the database server to skip all unavailable fragments. Use this option with caution. If a dbspace becomes unavailable, all queries initiated by applications that do not issue a SET DATASKIP OFF statement before they execute could be subject to errors.

Monitoring Fragmentation Use

The database administrator might find the following aspects of fragmentation useful to monitor:

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

The administrator can monitor the distribution of data over table fragments. If the goal of fragmentation is improved single-user response time, it is important for data to be distributed evenly over the fragments. To monitor fragmentation disk use, you must monitor database server tblspaces because the unit of disk storage for a fragment is a tblspace. (For information on how to monitor the data distribution for a fragmented table, see "Monitoring Tblspaces and Extents" on page 16-38.)

The administrator must monitor I/O request queues for data that is contained in fragments. When I/O queues become unbalanced, the administrator should work with the DBA to tune the fragmentation strategy. (For a discussion of how to monitor chunk use, including the I/O queues for each chunk, see "Monitoring Chunks" on page 16-34.)

The administrator must monitor fragments for availability and take appropriate steps when a dbspace that contains one or more fragments fails. For how to determine if a chunk is down, see "Monitoring Chunks" on page 16-34.

Displaying Databases

You can display databases that you create with SMI tables.

Using SMI Tables

Query the **sysdatabases** table to display a row for each database managed by the database server. For a description of the columns in this table, see the **sysdatabases** information in the chapter about the **sysmaster** database in the *Administrator's Reference*.

Monitoring the Database Server for Disabling I/O Errors

The database server notifies you about disabling I/O errors in two ways: the message log and event alarms.

Using the Message Log to Monitor Disabling I/O Errors

The database server sends the following message to the message log when a disabling I/O error occurs:

```
Assert Failed: Chunk {chunk-number} is being taken OFFLINE.
Who: Description of user/session/thread running at the time
Result: State of the affected database server entity
Action: What action the database server administrator should
take
See Also: DUMPDIR/af.uniqid containing more diagnostics
```

The result and action depend on the current setting of ONDBSPDOWN, as described in the following table.

ONDBSPDOWN Setting	Result	Action
`0`	Dbspace {*space-name*} is disabled.	Restore dbspace {*space-name*}.
`1`	The database server must abort.	Reinitialize shared memory.
`2`	The database server blocks at next checkpoint.	Use **onmode -k** to shut down, or use **onmode -O** to override.

For more information about interpreting messages that the database server sends to the message log, see the chapter about message-log messages in the *Administrator's Reference*.

Using Event Alarms to Monitor Disabling I/O Errors

When a dbspace incurs a disabling I/O error, the database server passes the following values as parameters to your event-alarm executable file.

Parameter	Value
Severity	`4 (Emergency)`
Class	`5`
Class message	`Dbspace is disabled: 'dbspace-name'`
Specific message	`Chunk {chunk-number} is being taken OFFLINE.`

If you want the database server to use event alarms to notify you about disabling I/O errors, write a script that the database server executes when it detects a disabling I/O error. For information about how to set up this executable file that you write, see the appendix on event alarms and the chapter on configuration parameters in the *Administrator's Reference*.

Monitoring Disk Usage

This section describes methods of tracking the disk space used by various database server storage units.

For background information about internal database server storage units mentioned in this section, see the chapter about disk structures and storage in the *Administrator's Reference*.

Monitoring Chunks

You can monitor chunks for the following information:

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

This information allows you to track the disk space used by chunks, monitor chunk I/O activity, and check for fragmentation.

Using Command-Line Utilities

You can use the following command-line utilities to obtain information about chunks.

onstat -d

The **onstat -d** utility lists all dbspaces and the following information for the chunks within those spaces.

- The address of the chunk
- The chunk number and associated dbspace number
- The offset into the device (in pages)
- The size of the chunk (in pages)
- The number of free pages in the chunk
- The pathname of the physical device
- Whether to skip logical replay for the dbspace

The dbspace flags indicate whether a dbspace is mirrored. The chunk flags provide the following information:

- Whether the chunk is the primary chunk or the mirrored chunk
- Whether the chunk is on-line, is down, is being recovered, or is a new chunk

***Important:** You must perform a level-0 backup of the root dbspace and the modified dbspace before mirroring can become active, and after turning off mirroring.*

Sample output for **onstat -D**, which displays the same information plus two additional fields, appears in Figure 16-2 on page 16-35. For descriptions of the **onstat -d** flags, see the utilities chapter in the *Administrator's Reference*.

onstat -D

The **onstat -D** option displays the same information as **onstat -d**, plus the number of pages read from the chunk (in the **page Rd** field).

Figure 16-2 shows sample output.

Figure 16-2
onstat -D Output

```
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
40d100   1        1        1        1        N        informix rootdbs
40d144   2        2        2        1        M        informix cookedspace
40d188   3        10       3        1        N B      informix cookedblob
 3 active, 10 total

Chunks
address  chk/dbs offset   page Rd  page Wr  pathname
40c274   1   1   0        146      4        /home/server/root_chunk
40c30c   2   2   0        1        0        /home/server/test_chunk
40c8fc   2   2   0        36       0        /home/server/test_mirr
40c3a4   3   3   0        4        0        /home/server/blob_chunk
 3 active, 10 total
```

onstat -g iof

The **onstat -g iof** option displays the number of reads from each chunk and the number of writes to each chunk. If one chunk has a disproportionate amount of I/O activity against it, this chunk might be a system bottleneck. This option is useful for monitoring the distribution of I/O requests against the different fragments of a fragmented table. Figure 16-3 shows sample output.

```
...
AIO global files:
gfd pathname          totalops  dskread dskwrite  io/s
  3 raw_chunk            38808    27241    11567   6.7
  4 cooked_chk1           7925     5660     2265   1.4
  5 cooked_chk2           3729     2622     1107   0.6
```

Figure 16-3
onstat -g iof Output

onutil CHECK RESERVED

To list the contents of the reserve pages, execute **onutil** CHECK RESERVED.

onutil CHECK SPACE

To obtain the physical layout of information, execute **onutil** CHECK SPACE.

The following information is displayed:

- The name, owner, and creation date of the dbspace
- The size in pages of the chunk, the number of pages used, and the number of pages free
- A listing of all the tables in the chunk, with the initial page number and the length of the table in pages

The tables within a chunk are listed sequentially. This output is useful for determining the extent of chunk fragmentation. If the database server is unable to allocate an extent in a chunk despite an adequate number of free pages, the chunk might be badly fragmented.

Using SMI Tables

Query the **syschunks** table to obtain the status of a chunk. The following columns are relevant.

Column	Description
chknum	Number of the chunk within the dbspace
dbsnum	Number of the dbspace
chksize	Total size of the chunk in pages
nfree	Number of pages that are free
is_offline	Whether the chunk is down
is_recovering	Whether the chunk is recovering
mis_offline	Whether the mirrored chunk is down
mis_recovering	Whether the mirrored chunk is being recovered

The **syschkio** table contains the following columns.

Column	Description
pagesread	Number of pages read from the chunk
pageswritten	Number of pages written to the chunk

Monitoring Tblspaces and Extents

Monitor tblspaces and extents to determine disk usage by database, table, or table fragment. Monitoring disk usage by table is particularly important when you are using table fragmentation, and you want to ensure that table data and table index data are distributed appropriately over the fragments.

Using Command-Line Utilities

You can use the following command-line utilities to monitor tblspaces and extents.

onutil CHECK TABLE INFO

Execute **onutil** CHECK TABLE INFO to obtain extent information.

You can include a database-name or table-name parameter with the command. The command displays the following information:

- Number of extents
- Size of the first extent
- Size of the next extent
- Number of pages allocated
- Number of pages used

Figure 16-4 shows sample output. The table in the example is fragmented over multiple dbspaces. Because each fragment of a fragmented table resides in a separate tblspace, the **onutil** CHECK TABLE INFO option always displays separate information for each fragment. The number of pages of table data in each fragment is displayed.

Figure 16-4
onutil CHECK TABLE INFO Output

```
TBLspace Report for tpc:informix.account

                  Table fragment in DBspace rootdbs

    Physical Address               100033
    Creation date                  03/31/99 13:25:21
    TBLspace Flags                 2          Row Locking
    Maximum row size               100
    Number of special columns      0
    Number of keys                 0
    Number of extents              2
    Current serial value           1
    First extent size              50
    Next extent size               25
    Number of pages allocated      2375
    Number of pages used           2370
    Number of data pages           2369
    Number of rows                 45001
    Partition partnum              2097154
    Partition lockid               2097154

    Extents
         Logical Page  Physical Page        Size
                   0        100ad5            50
                  50        100b2f          2325

                  Table fragment in DBspace dbspace2

    Physical Address               200005
    Creation date                  03/31/99 13:25:21
    TBLspace Flags                 2          Row Locking
    Maximum row size               100
    Number of special columns      0
    Number of keys                 0
    Number of extents              1
    Current serial value           1
    First extent size              50
    Next extent size               25
    Number of pages allocated      550
    Number of pages used           528
    Number of data pages           527
    Number of rows                 10000
    Partition partnum              3145730
    Partition lockid               2097154

    Extents
         Logical Page  Physical Page        Size
                   0        200035           550
...
```

onutil CHECK TABLE ALLOCATION INFO

The **onutil** CHECK TABLE ALLOCATION INFO returns all of the information from the **onutil** CHECK TABLE INFO option as well as additional information.

Figure 16-5 shows sample output. Each tblspace in the database or table that you supply is listed.

```
TBLSpace Usage Report for tpc:chrisw.account

    Type                  Pages      Empty  Semi-Full       Full  Very-Full
    ---------------- ---------- ---------- ---------- ---------- ----------
    Free                     20
    Bit-Map                   1
    Index                   471
    Data (Home)            3158
                     ----------
    Total Pages            3650

    Unused Space Summary

        Unused data slots                            2
        Unused bytes per data page                  44
        Total unused bytes in data pages        138952

Index Usage Report for index iaccount on tpc:chrisw.account

                     Average    Average
    Level    Total No. Keys Free Bytes
    ----- -------- -------- ----------
        1        1        4       1973
        2        4      116        506
        3      466      128        217
    ----- -------- -------- ----------
    Total      471      128        223
```

Figure 16-5 *Additional Information shown by onutil CHECK TABLE ALLOCATION INFO*

Using SMI Tables

Query the **systabnames** table to obtain information about each tblspace. The **systabnames** table has columns that indicate the corresponding table, database, and table owner for each tblspace.

Query the **sysextents** table to obtain information about each extent. The **sysextents** table has columns that indicate the database and the table that the extent belongs to, as well as the physical address and size of the extent.

Using System Catalog Tables

Query the **sysfragments** table to obtain information about all tblspaces that hold a fragment. This table has a row for each tblspace that holds a table fragment or an index fragment. The **sysfragments** table includes the following columns.

Column	Description
fragtype	Table or index fragment
tabid	Table identifier
indexname	Index identifier
partn	Physical location (tblspace ID)
strategy	Distribution scheme (round-robin, expression, table-based index)
dbspace	Dbspacename for fragment
npused	Number of data pages or leaf pages
nrows	Number of rows or unique keys

Not all columns of sysfragments are documented in the preceding list. For a complete listing of columns, see the *Informix Guide to SQL: Reference*.

Monitoring Simple Large Objects in a Dbspace

You can monitor dbspaces to determine the number of dbspace pages that TEXT and BYTE data use.

This command takes a database name or a table name as a parameter. For each table in the database, or for the specified table, the database server displays a general tblspace report.

Following the general report is a detailed breakdown of page use in the extent, by page type. See the **Type** column for information on TEXT and BYTE data.

The database server can store more than one simple large object on the same dbspace. Therefore, you can count the number of pages that store TEXT or BYTE data in the tblspace, but you cannot estimate the number of simple large objects in the table.

To view statistics for simple large objects, execute **onutil** CHECK TABLE ALLOCATION INFO. Figure 16-4 shows sample output.

No Compression of TEXT and BYTE Data Types

The database server does not contain any mechanisms for compressing TEXT and BYTE data after the data has been scanned into a database.

The database server scans TEXT and BYTE data into an existing table in the following ways.

Method to Scan TEXT or BYTE Data	Reference
DB-Access LOAD statement	LOAD statement in the *Informix Guide to SQL: Syntax*
dbload utility	*Informix Migration Guide*
Informix ESQL/C programs	*Informix ESQL/C Programmer's Manual*
From external tables	Chapter on loading with external tables in the *Administrator's Reference*

Chapter

Table Fragmentation and PDQ

17

In This Chapter

This chapter provides an overview of the table fragmentation and parallel database query (PDQ) features of your database server.

Table fragmentation allows you to store the parts of a table on different disks. Table fragmentation allows you to store large amounts of data in a single table and to balance the workload of large queries and high-transaction volumes across multiple disks.

Parallel 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 one aspect of a query among several processors.and coservers. 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 memory-resource management.

PDQ delivers maximum performance benefits when the data that is being queried is in fragmented tables. For information on how to use PDQ and fragmentation for maximum performance, refer to your *Performance Guide*.

Fragmentation

Fragmentation is a database server feature that enables you to define groups of rows or index keys within a table according to some algorithm or *scheme*. You can store each group or *fragment* in a separate dbspace that is associated with a specific physical disk. You create the fragments and assign them to dbspaces with SQL statements.

From the perspective of an end user or client application, a fragmented table is identical to a nonfragmented table. Client applications do not require any modifications to allow them to access the data that is contained in fragmented tables.

The database server stores the location of each table and index fragment, along with other related information, in the system catalog table named **sysfragments**. You can use this table to access information about your fragmented tables and indexes. For the complete listing of the information in this system catalog table, refer to the *Informix Guide to SQL: Reference*.

Because the database server has information on which fragments contain which data, the database server can route client requests for data to the appropriate fragment without accessing irrelevant fragments, as Figure 17-1 illustrates. For more information on fragment elimination, refer to your *Performance Guide*.

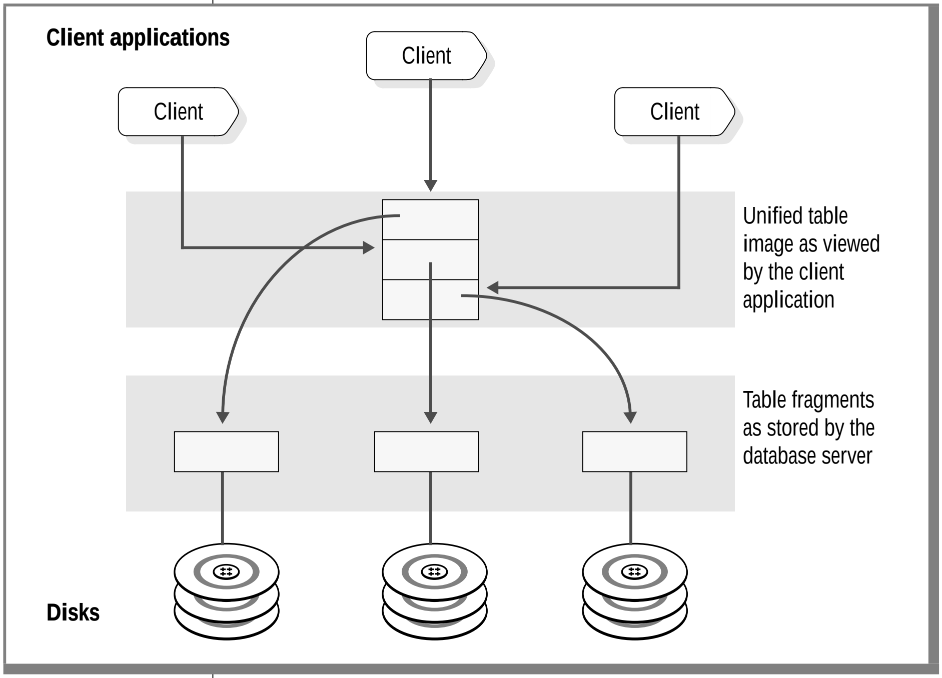

Client applications
Client
Client
Client
Unified table image as viewed by the client application
Table fragments as stored by the database server
Disks

Figure 17-1
Routing Client Requests To The Appropriate Table Fragments

The following sections cover these fragmentation topics:

- Fragmentation goals
- Fragmentation strategies
- Summary of SQL statements for fragmentation

Fragmentation Goals

Consider fragmenting your tables if you have at least one of the following goals:

- Improved single-user response time

  To improve the performance of individual queries, use fragmentation with parallel database query (PDQ) to scan in parallel fragments that are spread across multiple disks.
- Improved concurrency

  Fragmentation can reduce contention for data that is located in large tables that are used by multiple queries and OLTP applications. Fragmentation reduces contention because each fragment resides on a separate I/O device, and the database server directs queries to the appropriate fragment.
- Improved availability

  If a fragment becomes unavailable, the database server can still access the remaining fragments.
- Improved data-load performance

  When the database server uses parallel inserts and external tables to load a table that is fragmented across multiple coservers, it allocates threads to light append the data into the fragments in parallel. For more information on this load method, refer to the chapter on loading with external tables in the *Administrator's Reference*.
- Improved ALTER FRAGMENT performance

  You can also use the ALTER FRAGMENT TABLE with the ATTACH clause to add data quickly to a very large table.
- Improved backup-and-restore characteristics

  Fragmentation gives you a finer backup-and-restore granularity. This granularity can reduce the time that is required for backup-and-restore operations. In addition, you can improve the performance of backup-and-restore operations if you use ON-Bar to perform these operations in parallel.

Each of the preceding goals has its own implications for the fragmentation strategy that you ultimately implement. "Fragmentation Strategies" on page 17-6 discusses these issues. Your primary fragmentation goal determines, or at least influences, how you implement your fragmentation strategy.

In deciding whether to use fragmentation to meet any of the preceding goals, keep in mind that fragmentation requires some additional administration and monitoring activity. For more information about fragmentation, refer to your *Performance Guide*.

Responsibility for Fragmentation

Some overlap exists between the responsibilities of the database server administrator and those of the DBA (database administrator) with respect to fragmentation. The DBA creates the database schema. This schema can include table fragmentation. The database server administrator, on the other hand, lays out the disk space and creates the dbspaces where the fragmented tables reside. Because these responsibilities cannot be performed in isolation, implementing fragmentation requires a cooperative effort between the database server administrator and the DBA.

Fragmentation Strategies

A fragmentation strategy consists of two parts:

- A distribution scheme

  The scheme that you use to group rows or index keys into fragments is called the *distribution scheme*. You specify the distribution scheme in the FRAGMENT BY clause of the CREATE TABLE, CREATE INDEX, OR ALTER FRAGMENT statement.

- The set of dbspaces in which you locate the fragments

  You specify the set of dbspaces in the IN clause of these SQL statements.

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, and the second and subsequent rows are assigned to fragments in sequence. If one of the fragments is full, that fragment is skipped.
- **Expression-based.** This type of fragmentation puts rows into fragments based on a *fragmentation expression* that you specify. This expression defines criteria, or rules, for assigning a set of rows to each fragment. The expression can take the form of a range or some other 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.
- **System-defined hash.** This type of fragmentation uses an internal, system-defined rule that distributes rows with the object of keeping the same number of rows in each fragment.
- **Hybrid fragmentation**. This type of fragmentation combines two types of distribution schemes to put rows into fragments in different dbslices and dbspaces. You specify an expression-based distribution scheme to choose a dbslice, and a system-defined hash distribution scheme to fragment the table across dbspaces within that dbslice.

Table Fragmentation

Formulating a fragmentation strategy for a table requires you to make the following decisions:

1. Decide what your primary fragmentation goal is.

   Your fragmentation goals 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.
4. To complete the fragmentation strategy, you must decide on the number and location of the fragments.

For more information on the decisions that you must make to formulate a fragmentation strategy, see the *Informix Guide to Database Design and Implementation*. For information on optimizing the performance of your fragmentation scheme, refer to your *Performance Guide*.

Temporary Table Fragmentation

Just as you fragment permanent tables, you also can fragment an explicit temporary table across multiple disks.

To create a temporary, fragmented table, use the TEMP or SCRATCH keyword of the CREATE TABLE statement.

You can specify what distribution scheme and which dbspaces to use for the temporary table. For more information on the types of temporary tables, refer to "Temporary Tables" on page 15-37.

You can define your own fragmentation strategy for an explicit temporary table, or you can let the database server dynamically determine the fragmentation strategy. For more information, refer to your *Performance Guide*.

Table Index Fragmentation

You can fragment both table data and table indexes. When you create an index, you can:

- create an *attached index* by omitting the storage specification from the CREATE INDEX statement.

  When you do, the attached index takes on the same fragmentation strategy as the table. Each fragment of an attached index resides in the same dbspace as the corresponding table data.

  You create an attached index by omitting the FRAGMENT BY and IN clauses from the CREATE INDEX statement.

  ```
  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);
  ```

- create a *detached index* by including an explicit storage specification in the CREATE INDEX statement.

  When you do, the detached index uses its own fragmentation strategy, which can differ from that of the table. A fragment in a detached index can reside in a different dbspace than the corresponding table data.

  You cannot use the round-robin distribution scheme for an index. For more information on the CREATE INDEX statement, refer to the *Informix Guide to SQL: Syntax*.

Fragmenting table data and table indexes can greatly affect performance. For detailed information on fragmenting table data and table indexes, see your *Performance Guide*.

SQL Statements That Perform Fragmentation Tasks

To perform most fragmentation tasks, you use appropriate SQL statements. Figure 17-2 lists the fragmentation tasks and the SQL statements to accomplish these tasks.

For details on how to accomplish these fragmentation tasks, refer to the *Informix Guide to Database Design and Implementation*. For the syntax of these SQL statements, refer to the *Informix Guide to SQL: Syntax*.

Figure 17-2
Fragmentation Tasks and Corresponding SQL Statements

Fragmentation Task	SQL Statements
Creating a new fragmented table	CREATE TABLE statement, FRAGMENT BY clause
Creating a fragmented table from a single nonfragmented table	ALTER FRAGMENT statement, INIT clause
Creating a fragmented table from more than one nonfragmented table	ALTER FRAGMENT statement, ATTACH clause
Modifying distribution scheme for a fragmented table	ALTER FRAGMENT statement, INIT clause
Adding a fragment to a table	ALTER FRAGMENT statement, ATTACH clause
Removing a fragment from a table	ALTER FRAGMENT statement, DETACH clause
Reinitializing a fragmentation scheme	ALTER FRAGMENT statement, INIT clause
Converting a fragmented table to a non-fragmented table	ALTER FRAGMENT statement, INIT clause
Creating a fragmented index	CREATE INDEX statement
Adding an explicit rowid column to a fragmented table	Not supported

Parallel Database Query

PDQ refers to the techniques that the database server can use to distribute the execution of a single query over several processors and coservers in Extended Parallel Server.

The database server can also use PDQ for queries that consume large quantities of non-CPU resources, in particular large quantities of memory and many disk scans.

A query that is processed with PDQ techniques is called a *PDQ query*. When the database server processes a PDQ query, it first divides the query into subplans. The database server then allocates the subplans to a number of threads that process the subplans in parallel. Because each subplan represents a smaller amount of processing time when compared to the original query, and because each subplan is processed simultaneously with all other subplans, the database server can drastically reduce the time that is required to process the query. Figure 17-3 illustrates this concept.

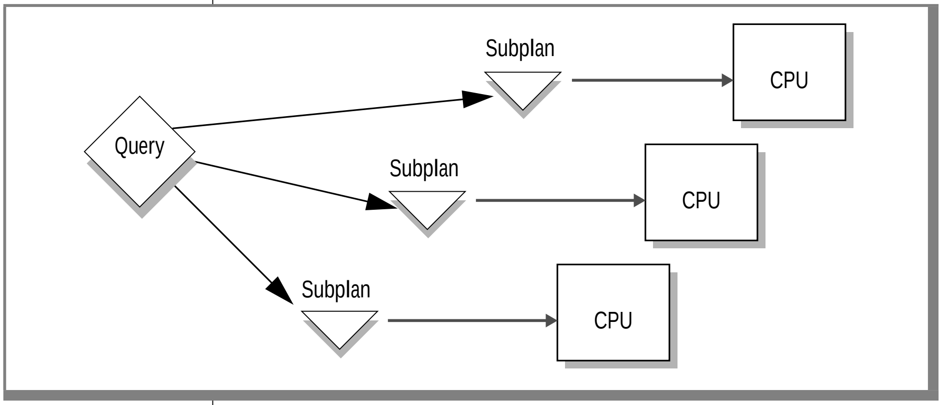


Figure 17-3
Parallel Database Query

Parallelism

The degree of parallelism for a query refers to the number of subplans that the database server executes in parallel to run the query. For example, a two-table join that six threads execute (with each thread executing one sixth of the required processing) has a higher degree of parallelism than one that two threads execute.

The database server determines the best degree of parallelism for each component of a PDQ query, based on various considerations: the number of available coservers, the number of virtual processors (VPs) on each coserver, the fragmentation of the tables that are being queried, the complexity of the query, and so forth.

The database server achieves a high degree of parallelism, so SQL operations are completely parallel. *Completely parallel* means that Extended Parallel Server processes multiple threads simultaneously on all CPU VPs across all coservers to speed execution of a single query.

The value of PDQPRIORITY does not determine when to use PDQ to process a query in parallel. Even when the value of PDQPRIORITY is 0, the database server executes a query in parallel across all CPU VPs on all coservers.

Important: *In Extended Parallel Server, PDQPRIORITY does not affect the degree of parallelism. PDQPRIORITY values that are set by the database server administrator, by the user, and by the client application affect only the amount of memory available for parallel processing.*

PDQ provides performance advantages on parallel-processing platforms composed of multiple computers.On a parallel-processing platform, PDQ distributes the execution of a query across available processors on all nodes that support coservers, and takes full advantage of the memory on each of those nodes.

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

Extended Parallel Server distributes each subplan to the pertinent coservers and executes the subplans in parallel. Each subplan is processed simultaneously with the others. Because each subplan represents a smaller amount of processing time than the original query plan, the database server can drastically reduce the time that is required to process the query if each portion of the query had to be performed consecutively.

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

For example, consider the following SQL request:

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

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

1. Each coserver scans relevant fragments of the **customer** table and the **cash** table in parallel.
2. Each coserver joins rows from local fragments of both the **customer** table and the **cash** table by customer ID.
3. As participating coservers complete local join operations, they can go on to perform other portions of the join operation or aggregations. They can also perform some of the steps that are involved in selecting the geographic areas and dollar amounts that belong to particular customers, the group-by operations, and the order-by operations that are needed to complete the query.
4. When the query is complete, the connection coserver returns the results to the client.

Structure of a PDQ Query

The database server divides a query into components that can be performed in parallel to increase the speed of query execution significantly.

Depending on the number of tables or fragments that a query must search, the optimizer determines if a query subplan can execute in parallel.

The Resource Grant Manager (RGM) assigns the different components of a query to different threads across processors on different coservers.

The **sqlexec** thread initiates these component threads, which the SET EXPLAIN output lists as *secondary threads*.

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 it does so, the join thread is considered a producer, and the sort thread is considered a consumer.

The database server uses *SQL operators* and *exchanges* to divide a query plan into subplans that can be performed in parallel to increase the speed of query execution significantly.

SQL Operators

An *SQL operator* is a process that accepts a stream of rows from one or two data tables. Each SQL operator reads each row in a stream and applies a predefined behavior to the data.

For example, a typical query plan might contain scan and hash join SQL operators. The behavior of these SQL operators is as follows:

- Scan

  This type of SQL operator performs a sequential read on:

  - a table or index fragment.
  - an unfragmented table or index.

  The scan SQL operators handle data from a *local table* or *local index*. A local table or local index resides on the same coserver on which the SQL operator executes.

- Hash join

  This type of SQL uses a hash method to join tables. It selects one table from to constructs a hash table. It then uses that hash table to join data from other tables involved in the join operation.

The database server creates multiple instances of each SQL operator to execute on different parts of the data in parallel, as follows:

- The scan operators execute in parallel, based on the fragmentation strategy of the tables.
- The hash join operators execute in parallel, based on the availability of resources (such as memory and number of CPU VPs) on the coservers.

The database server structures queries into a plan of SQL operators. Figure 17-4 on page 17-16 shows the SQL operator plan that the database server constructs to process the following SQL query:

```
SELECT geo_id, dollars
    FROM customer a, cash b
    WHERE a.cust_id=b.cust_id;
```

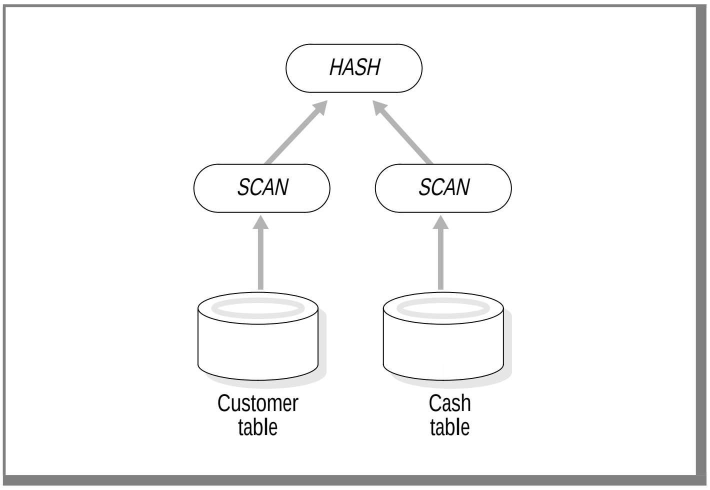


Figure 17-4
SQL Operator Plan for a Query

Exchanges

An *exchange* is another process that affects parallel processing. An exchange takes the results of two or more instances of an SQL operator and initiates another set of operators to process the next SQL operator that is required to complete the query. The database server inserts exchanges at places within an SQL operator plan where parallelism is beneficial.

When several instances of an SQL operator supply data to another SQL operator, the exchange synchronizes the transfer of data from the multiple instances to the next SQL operator. For instance, if two fragmented tables are to be joined, the optimizer typically calls for a separate scan thread for each fragment. Because of different I/O characteristics, the scan threads can complete their work at different times. The database server uses an exchange to funnel the data that the various scan threads produce into one or more join threads with a minimum amount of buffering.

PDQ Threads

Depending on the resources that are available for a decision-support query, the database server assigns the different components of a query plan to different threads across coservers. The **sqlexec** thread initiates these PDQ threads, which the SET EXPLAIN output lists as *secondary threads*.

The database server creates these secondary threads and exchanges automatically and transparently. They are terminated automatically as they complete processing for a given query. The database server creates new threads and exchanges as needed for subsequent queries.

Some monitoring tools display only the SQL operator but not the exchanges. For more information on monitoring PDQ, refer to your *Performance Guide*.

Use of PDQ

Applications that access data stored in a relational database can be divided into the following two types:

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

The complex queries that are typical of decision-support applications can benefit from PDQ.

The next sections describe the characteristics of OLTP and decision-support applications.

OLTP Applications

OLTP applications are characterized by quick, indexed access to a small number of data items. An order-entry system is an example of a typical OLTP system. The transactions handled by OLTP applications are usually simple and predefined.

OLTP applications can be characterized as follows:

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

The default behavior of the database server is ideal for OLTP transactions, optimizing performance for short transactions that require rapid response times. All queries have the same priority for CPU, memory, and disk I/O.

Queries that require quick response and generate only a small amount of information should *not* use PDQ. For example, the following queries should not use PDQ:

- Do we have a hotel room available in Berlin on December 8?
- Does the store in Mill Valley have green tennis shoes in size 4?

The impact of PDQ on OLTP queries can be dramatic. One PDQ parameter limits the number of simultaneous queries the database server can perform. Suppose the number of simultaneous queries is set to 4. If another query requests service, it must wait until one of the previous four queries finishes. If the four queries are decision-support queries, the delay could be several minutes. Typical OLTP queries must be processed immediately.

Decision-Support Applications

Decision-support applications provide information for strategic planning, decision making, and report preparation. Decision-support applications frequently generate queries that require the database server to scan entire tables and manipulate large amounts of data. These queries can require operations such as multiple joins, temporary tables, and hundreds, if not thousands, of calculations. For example, the following queries should use the PDQ features of the database server:

- Based on the predicted number of housing starts, the known age of existing houses, and the observed roofing choices for houses in different areas and price ranges, what roofing materials should we order for each of our regional distribution centers?
- How does the cost of health-care plan X compare with the cost of health-care plan Y, considering the demographic profile of our company? Would plan X be better for some regions and plan Y for others?

Such operations require large amounts of data and large amounts of memory. As a result, the execution times for decision-support applications are far longer than the execution times required for typical OLTP applications. Other typical decision-support applications include payroll, inventory reporting, and end-of-period accounting reports. These applications are frequently executed in a batch environment.

Queries that contain one or more of the following operations require large quantities of memory:

- Hash joins
- Sorting
- Groups

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

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

If no indexes exist on **table1**, a sort is required, and hence the database server must allocate memory and temporary disk space to sort the query. However, if column **col1** is indexed, the query does not require these resources.

Decision-support applications have the following characteristics:

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

When both OLTP and decision-support queries are running on the same computer, the database server must balance its resources so that all users receive the best possible performance.

Database Server Use of PDQ

In Extended Parallel Server, parallel execution automatically occurs when the database operation involves data that is fragmented across multiple dbspaces and multiple CPU VPs are available. Parallel processing can occur on both a single coserver and across multiple coservers.

- Single coserver execution

  Extended Parallel Server executes database operations in parallel when:

  - the involved tables are fragmented across separate dbspaces on separate disks that are local to one coserver.
  - an SQL operator in the query plan processes a large amount of data so that it dynamically allocates multiple threads to execute in parallel across available CPU VPs on the (single) local coserver.
- Multiple coserver execution

  Extended Parallel Server achieves a high degree of parallelism for a query when the involved tables are fragmented across all coservers and multiple CPU VPs on multiple coservers execute the SQL operators within the query plan.

Important: *In Extended Parallel Server, the value of PDQPRIORITY does not determine when to use PDQ to process a query in parallel. The database server does not use PDQ if the database operation accesses data on only one table fragment on a single coserver or does not contain an SQL operator that can execute in parallel if multiple CPU VPs are available.*

Extended Parallel Server executes the following database operations in parallel when the involved tables are fragmented into separate dbspaces:

- DSS queries, which are usually complex SELECT statements that involve a large number of rows to scan, join, aggregate, sort, or group
- INSERT, DELETE, and UPDATE statements that process nonlocal data
- SPL routines
- Sorts
- Index builds
- Update statistics
- **onutil** commands, such as CHECK DATA and CHECK INDEX command options
- Uncorrelated subqueries
- Correlated subqueries if they cannot be unnested

For more information on how Extended Parallel Server uses PDQ to execute these database operations, refer to your *Performance Guide*.

Resource Grant Manager

The Resource Grant Manager (RGM) is a database server component that coordinates the use of resources, where parallel branches are executed, and which queries to run if more than one query is in the queue.

The RGM dynamically allocates the following resources for PDQ queries and other parallel database operations:

- The amount of memory in the virtual portion of database server shared memory that the query can reserve

  The RGM uses configuration parameters, environment variables, and SQL statements to determine how to grant memory to a decision-support query.

- The number of parallel threads that can be started for each query

  The RGM uses the SQL operators that make up the query plan to determine the number of threads to start for a query.

  For join, group, and sort SQL operators, the RGM uses the following factors to determine the number of threads to start:

  - The values of configuration parameters (NUMCPUVPS, DS_TOTAL_MEMORY, DBSPACETEMP and so forth) that the database server administrator sets

    For more information on the parameters that you can set to affect parallelism, refer to your *Performance Guide*.
  - The number of CPU VPs available
  - The availability of computer-system resources (CPUs, memory, and disk I/O)

  For scan and insert SQL operators, the RGM also uses the following factor to determine the number of scan and insert threads (disk I/O operators):

  - The number of fragments that the database operation accesses

For information about RGM, SQL operators, and fragmentation strategy guidelines to improve performance, refer to your *Performance Guide*.

Fragmentation Enhancement to PDQ

The complex queries that are typical of DSS applications benefit from PDQ and fragmentation. DSS applications perform complex tasks that often include scans of entire tables, manipulation of large amounts of data, multiple joins, and the creation of temporary tables. Such operations can involve many I/O operations, many calculations, and large amounts of memory.

The performance of decision-support queries increases almost linearly with the number of fragments added. With Extended Parallel Server, you can add many more fragments based on the number of parallel nodes with attached disks on separate I/O ports. Informix recommends that the number of fragments be a multiple of the number of coservers.

When you fragment your data across multiple coservers, the database server can start parallel SQL operators (scans, sorts, inserts, and so forth) on all available CPU virtual processors on the different coservers.

In Extended Parallel Server, you can increase the degree of parallelism by fragmenting tables across multiple coservers. Cross-coserver fragmentation ensures that table fragments are processed in parallel by threads running on each coserver.

Fragmenting tables across coservers provides these advantages:

- More efficient use of shared memory

  The database server uses the resources on each coserver when fragments are processed in parallel.
- More efficient hash algorithm

  The system-defined hash fragmentation rule lets the database server eliminate fragments immediately for queries that use the hashed column as a join key.
- Higher degree of parallelism for scans

  The separate disks and CPU VPs on each coserver can process I/O in parallel.
- More efficient join operations

  Co-located joins generate less traffic between coservers.
- Higher degree of parallelism for sorts

  If the data involved in the sort operation resides on different coservers, parallel sorts can occur even when the **PSORT_NPROCS** environment variable is not set.

For more information on fragmentation strategy guidelines to improve performance, refer to your *Performance Guide*.

How the Database Server Balances Workload

Data skew is a condition that occurs when the majority of the data involved in an SQL operation resides on one coserver and is not distributed evenly across multiple coservers.

For example, during a hash join, data skew occurs when a large number of duplicate values exist in the join column. The bulk of the data values involved in the hash join reside on one or two of the coservers. As a result, one or two of the coservers are still processing rows for the hash join, while the other coservers have completed and are waiting for the rest of the rows before the next SQL operator can start processing.

RGM can detect data skew during the build or probe phase of the hash join. The database server detects if one coserver has many more rows to process than the other coservers. If data skew occurs, RGM distributes the data from the coserver with the most rows to the other idle coservers to perform part of hash join.

The database server creates a new SQL operator, the flex join operator, to process the redistributed hash join rows on the other coservers.

For more information on the features that Extended Parallel Server provides to balance the workload, refer to your *Performance Guide*.

Resource Allocation with PDQ

This section discusses the configuration parameters, environment variables, and SELECT statements that you can use to balance resource use.

Parameters for Controlling PDQ

Figure 17-5 summarizes the configuration parameters, environment variables, and the SQL statement that control how the database server allocates resources to PDQ in Extended Parallel Server. The value set by the SQL statement supersedes values set by the environment variables, and values set by environment variables supersede values set by configuration parameters.

Figure 17-5
Parameters Used for Controlling PDQ in Extended Parallel Server

Configuration Parameters	Environment Variables	SQL Statements	Purpose of Parameter
DS_ADM_POLICY			Indicate policy that RGM should use to schedule queries.
DS_MAX_QUERIES			Maximum number of PDQ queries that can be active at any one time
MAX_PDQPRIORITY			Percentage of user's requested PDQPRIORITY value that the database server grants
OPTCOMPIND	**OPTCOMPIND**		Indicate a preferred join type to the query optimizer
PDQPRIORITY	**PDQPRIORITY**	SET PDQPRIORITY	Request minimum and maximum percentage of PDQ memory for an application or a specific query
		SET SCHEDULE LEVEL	Request scheduling priority for an application or a specific query

For more information on the PDQ parameters, refer to the chapter on configuration parameters in the *Administrator's Reference* and to your *Performance Guide*.

Section V

Logging and Log Administration

Chapter

Logging

18

In This Chapter

This chapter describes logging of Informix database server functions and addresses the following questions:

- Which database server processes require logging?
- What is transaction logging?
- What database server activity is logged?
- What are logging and nonlogging tables?

Next, the chapter describes logging of databases and addresses the following questions:

- What is the database-logging status?
- Who can set or change the database logging status?

Database Server Processes That Require Logging

As the Informix database server operates—as it processes transactions, keeps track of data storage, ensures data consistency, and so on—it automatically generates *logical-log records* for some of the actions that it takes. Most of the time, the database server makes no further use of the logical-log records. However, when the database server needs to roll back a transaction, to execute a fast recovery after a system failure, for example, the logical-log records are critical. The logical-log records are at the heart of the data-recovery mechanisms.

The database server stores the logical-log records in a *logical log*. The logical log is made up of *logical-log files* that the database server manages on disk until they have been safely transferred off-line (*backed up*). The database server administrator keeps the off-line logical-log records (in the backed-up logical-log files) until they are needed during a data restore, or until the administrator decides that the records are no longer needed for a restore. For more information, see Chapter 20, "Logical Log."

The database server uses logical-log records when it performs various functions that recover data and ensure data consistency, as follows:

- Transaction rollback

  If a database is using transaction logging and a transaction must be rolled back, the database server uses the logical-log records to reverse the changes made during the transaction. For more information, see "Transaction Logging" on page 18-5.

- Fast recovery

  If the database server shuts down in an uncontrolled manner, the database server uses the logical-log records to recover all transactions that occurred since the oldest update not yet flushed to disk and to roll back any uncommitted transactions. (When all the data in shared memory and on disk are the same, they are *physically consistent*.) The database server uses the logical-log records in the second phase of fast recovery when it returns the entire database server to a state of logical consistency up to the point of the most-recent logical-log record. (For more information, see "Details of Fast Recovery After A Full Checkpoint" on page 24-14.)

- Data restoration

  During a data restore, you use the logical-log backup with the most-recent storage-space backup to re-create the database server system up to the point of the most recently backed-up logical-log record. After restoring the storage spaces, the database server restores the logical logs to reimplement all the logged activity since the last storage-space backup.

- Deferred checking

  If a transaction uses the SET CONSTRAINTS statement to set checking to DEFERRED, the database server does not check the constraints until the transaction is committed. If a constraint error occurs while the transaction is being committed, the database server uses logical-log records from the transaction to roll back the transaction. For more information, see SET Database Object Mode in the *Informix Guide to SQL: Syntax*.

- Cascading deletes

  Cascading deletes on referential constraints use logical-log records to ensure that a transaction can be rolled back if a parent row is deleted and the system fails before the children rows are deleted. For information on table inheritance, see the *Informix Guide to Database Design and Implementation*. For information on primary key and foreign key constraints, see the *Informix Guide to SQL: Tutorial*.

Transaction Logging

A database or table is said to *have* or *use* transaction logging when SQL data manipulation statements in a database generate logical-log records.

The database-logging *status* indicates whether a database uses transaction logging. The *log-buffering mode* indicates whether a database uses buffered or unbuffered logging, or ANSI-compliant logging. For more information, see "Database-Logging Status" on page 18-9 and Chapter 19, "Managing Database-Logging Status."

In Extended Parallel Server, databases *always* use transaction logging. If you do not specify the buffering mode for a database, the default is unbuffered logging. You can use the SQL statement SET LOG or the **ondblog** utility to change the log-buffering mode. For information on **ondblog**, see the chapter on utilities in the *Administrator's Reference*.

Although databases are always logged, you can use logging or nonlogging tables within a database. The user who creates the table specifies the type of table. Even if you use nonlogging tables, the database server always logs some events. For more information, see "Logging and Nonlogging Tables" on page 18-8.

Transactions against multiple coservers are always unbuffered. If any regular table or fragment involved in a transaction resides on a coserver other than the connection coserver for the application that makes the request, the database server uses unbuffered logging for that transaction. If all tables involved in the transaction are raw tables, no buffering takes place.

For local transactions, you can specify either buffered or unbuffered logging. (Local transactions are operations on the same coserver, including connection to applications). The table type determines the log-buffering mode for local transactions on one coserver.

Database Server Activity That Is Logged

The database server does not generate logical-log records for every operation because it does not need a record of every action. The database server needs logical-log records only to perform the functions listed in "Database Server Processes That Require Logging" on page 18-3. Also, the space required to store a record of everything that the database server did would quickly become unwieldy.

The logical-log records themselves are variable length. This arrangement increases the number of logical-log records that can be written to a page in the logical-log buffer. However, the database server often flushes the logical-log buffer before the page is full.

Two types of logged activity are possible in the database server:

- Activity that is always logged
- Activity that is logged only for databases that use transaction logging

The following sections explain the two different types of activity. For more information on the format of logical-log records, see the chapter on interpreting logical-log records in the *Administrator's Reference*.

Activity That Is Always Logged

Some database operations always generate logical-log records, even if you use nonlogging tables on Extended Parallel Server.

The following operations are always logged for permanent tables:

- SQL data definition statements:

ALTER INDEX	CREATE VIEW
ALTER TABLE	DROP INDEX
CREATE DATABASE	DROP PROCEDURE
CREATE INDEX	DROP SYNONYM
CREATE PROCEDURE	DROP TABLE
CREATE SCHEMA	DROP TRIGGER
CREATE SYNONYM	DROP VIEW
CREATE TABLE	RENAME COLUMN
CREATE TRIGGER	RENAME TABLE

- Storage-space backups
- Checkpoints
- Administrative changes to the database server configuration such as adding a chunk or dbspace
- Allocation of new extents to tables
- A change to the logging status of a database

Activity Logged for Databases with Transaction Logging

If a database uses transaction logging, all SQL data manipulation statements, except SELECT, against that database generate one or more log records. These statements are as follows:

- DELETE
- INSERT
- LOAD
- SELECT INTO TEMP
- UNLOAD
- UPDATE

If these statements are rolled back, the rollback also generates log records.

Logging and Nonlogging Tables

Data warehousing and similar applications that involve very large amounts of data and few or no inserts, updates, or deletes often need a mix of logged and nonlogged tables within the same database. Extended Parallel Server supports both logging and nonlogging tables. These tables can be permanent or temporary.

STANDARD, OPERATIONAL, and TEMP tables are logging tables while STATIC, RAW, and SCRATCH tables are nonlogging tables. The following are some guidelines for choosing a table type:

- Choose STANDARD tables for OLTP operations.
- Choose OPERATIONAL tables when loading data from another database, or if you need to log transactions but do not care about restorability.
- Choose STATIC tables for data that rarely changes.
- Choose RAW tables when loading data from external tables.
- Choose temporary tables to reduce sorting scope, select an ordered subset of table rows, or to copy tables.

  Whether you use SCRATCH or TEMP tables depends on whether you need logging and indexing. If you do not need to rollback data in temporary tables, you can achieve maximum performance by using SCRATCH tables in temporary dbspaces.

For more information on the different table types, see "Table Types" on page 15-25 and your *Performance Guide*.

To switch from one table type to another, use the ALTER TABLE command. For more information, refer to the *Informix Guide to SQL: Syntax* and "Modifying the Table-Logging Status" on page 19-6.

Use of Logging Tables

Use logging tables if users are updating the data frequently or the ability to recover any updated data is critical. You must use logging tables if users are executing data transactions.

Use of Nonlogging Tables

Choose nonlogging tables if users are primarily analyzing the data and updating it infrequently. Back up nonlogging tables to ensure that you can restore them if transactions or the database server should fail. When you use RAW or SCRATCH tables, data consistency is not guaranteed when moving rows from one fragment to another if an error occurs during an update of the fragmentation columns.

Activity That Is Not Logged

For temp tables in temporary dbspaces, nothing is logged, not even the SQL statements listed in "Activity That Is Always Logged" on page 18-7. If you include temporary (nonlogging) dbspaces in DBSPACETEMP, the database server places nonlogging tables in these temporary dbspaces first. For more information, see "Logging and Nonlogging Tables" on page 18-8.

Database-Logging Status

You must use transaction logging with a database to take advantage of any of the features listed in "Database Server Processes That Require Logging" on page 18-3.

Every database that the database server manages has a logging status. The logging status indicates whether the database uses transaction logging and, if so, which log-buffering mechanism the database employs. To find out the transaction-logging status of a database, use the database server utilities, as explained in "Monitoring Transaction Logging" on page 19-7. The database-logging status indicates any of the following types of logging:

- Unbuffered transaction logging
- Buffered transaction logging
- ANSI-compliant transaction logging

All logical-log records pass through the logical-log buffer in shared memory before the database server writes them to the logical log on disk. However, the point at which the database server flushes the logical-log buffer is different for buffered transaction logging and unbuffered transaction logging. For more information, see "How the Database Server Uses Shared Memory" on page 13-6 and "Flushing the Logical-Log Buffer" on page 13-52.

Unbuffered Transaction Logging

If transactions are made against a database that uses unbuffered logging, the records in the logical-log buffer are guaranteed to be written to disk during commit processing. When control returns to the application after the COMMIT statement (and before the PREPARE statement for distributed transactions), the logical-log records are on the disk. The database server flushes the records as soon as any transaction in the buffer is committed (that is, a commit record is written to the logical-log buffer).

When the database server flushes the buffer, only the used pages are written to disk. Used pages include pages that are only partially full, however, so some space is wasted. For this reason, the logical-log files on disk fill up faster than if all the databases on the same database server use buffered logging.

Unbuffered logging is the best choice for most databases because it guarantees that all committed transactions can be recovered. In the event of a failure, only uncommitted transactions at the time of the failure are lost. However, with unbuffered logging, the database server flushes the logical-log buffer to disk more frequently, and the buffer contains many more partially full pages, so it fills the logical log faster than buffered logging does.

Buffered Transaction Logging

If transactions are made against a database that uses buffered logging, the records are held (*buffered*) in the logical-log buffer for as long as possible. They are not flushed from the logical-log buffer in shared memory to the logical log on disk until one of the following situations occurs:

- The buffer is full.
- A commit on a database with unbuffered logging flushes the buffer.
- A checkpoint occurs.
- The connection is closed.

If you use buffered logging, and a failure occurs, you cannot expect the database server to recover the transactions that were in the logical-log buffer when the failure occurred. Thus, you could lose some committed transactions. In return for this risk, performance during alterations improves slightly. Buffered logging is best for databases that are updated frequently (when the speed of updating is important), as long as you can re-create the updates in the event of failure. You can tune the size of the logical-log buffer to find an acceptable balance for your system between performance and the risk of losing transactions to system failure.

ANSI-Compliant Transaction Logging

The ANSI-compliant database logging status indicates that the database owner created this database using the MODE ANSI keywords. ANSI-compliant databases always use unbuffered transaction logging, enforcing the ANSI rules for transaction processing. You cannot change the buffering status of ANSI-compliant databases.

Databases with Different Log-Buffering Status

All databases on a database server use the same logical log and the same logical-log buffers. Therefore, transactions against databases with different log-buffering statuses can write to the same logical-log buffer. In that case, if transactions exist against databases with buffered logging *and* against databases with unbuffered logging, the database server flushes the buffer *either* when it is full *or* when transactions against the databases with unbuffered logging complete.

Settings or Changes for Logging Status or Mode

The user who creates a database with the CREATE DATABASE statement establishes the logging status or buffering mode for that database. For more information on the CREATE DATABASE statement, see the *Informix Guide to SQL: Syntax*.

Databases are always logged in Extended Parallel Server.

Only the database server administrator can change logging status. Chapter 19, "Managing Database-Logging Status," describes this topic. Ordinary end users cannot change database-logging status.

If a database does not use logging, you do not need to consider whether buffered or unbuffered logging is more appropriate. If you specify logging but do not specify the buffering mode for a database, the default is unbuffered logging.

End users *can* switch from unbuffered to buffered (but not ANSI-compliant) logging and from buffered to unbuffered logging for the *duration of a session*. The SET LOG statement performs this change within an application. For more information on the SET LOG statement, see the *Informix Guide to SQL: Syntax*.

Chapter

19 Managing Database-Logging Status

In This Chapter

This chapter covers the following topics on changing the database-logging status:

- Understanding database-logging status
- Modifying database-logging status with **ondblog**
- Monitoring transaction logging

As a database server administrator, you can alter the logging status of a database as follows:

- Change transaction logging from buffered to unbuffered.
- Change transaction logging from unbuffered to buffered.
- Make a database ANSI compliant.
- Change a table from logging to nonlogging.
- Change a table from nonlogging to logging.

For information about database-logging status, when to use transaction logging, and when to buffer transaction logging, see Chapter 18, "Logging." To find out the current logging status of a database, see "Monitoring Transaction Logging" on page 19-7.

Changing Database-Logging Status

You can use **ondblog** to change logging and then use ON-Bar to back up the data. For information on ON-Bar, see the *Backup and Restore Guide*.

Figure 19-1 shows how the database server administrator can change the database-logging status. Certain logging status changes take place immediately while other changes require a level-0 backup.

Figure 19-1
Logging Status Transitions

Converting from:	Converting to:			
	No Logging	**Unbuffered Logging**	**Buffered Logging**	**ANSI Compliant**
Unbuffered logging	Yes	Not applicable	Yes	Yes
Buffered logging	Yes	Yes	Not applicable	Yes
ANSI compliant	Illegal	Illegal	Illegal	Not applicable

Tip: *To change the logging status of ANSI-compliant databases, unload and reload the data. For more information, see "Making a Database ANSI Compliant with ondblog" on page 19-6.*

Some general points about changing the database-logging status follow:

- When you change the logging status, the database server places an exclusive lock on the database to prevent other users from accessing the database.
- If a failure occurs during a logging-mode change, check the logging mode in ON-Monitor or the flags in the **sysdatabases** table in the **sysmaster** database after you restore the database server data. For more information, see "Monitoring Transaction Logging" on page 19-7.

- Once you choose either buffered or unbuffered logging, an application can use the SQL statement SET LOG to change from one logging mode to the other. This change lasts for the duration of the session. For information on SET LOG, see the *Informix Guide to SQL: Syntax*.
- Databases *always* use transaction logging. You can specify what log-buffering mode they use. You also can turn logging on or off for tables.

Modifying Database-Logging Status with ondblog

You can use the **ondblog** utility to change the logging mode for one or more databases. If you add logging to a database, you must create a level-0 backup before the change takes effect. For more information, see the section on using **ondblog** in the *Administrator's Reference*.

Changing Buffering Status with ondblog

To change the buffering status from buffered to unbuffered logging on a database called **stores_demo**, execute the following command:

```
ondblog unbuf stores_demo
```

To change the buffering status from unbuffered to buffered logging on a database called **stores_demo**, execute the following command:

```
ondblog buf stores_demo
```

Canceling a Logging Mode Change with ondblog

To cancel the logging mode change request before the next level-0 backup occurs, execute the following command:

```
ondblog cancel stores_demo
```

Making a Database ANSI Compliant with ondblog

Once you convert a database to ANSI mode, you cannot change it to any other logging mode. To make a database called **stores_demo** into an ANSI-compliant database with **ondblog**, execute the following command:

```
ondblog ansi stores_demo
```

Modifying the Table-Logging Status

Extended Parallel Server creates standard tables that use logging by default. For more information, refer to "Logging and Nonlogging Tables" on page 18-8.

For more information on ALTER TABLE and SELECT, see the *Informix Guide to SQL: Syntax*.

Altering a Table to Turn Off Logging

To switch a table from logging to nonlogging, use the SQL statement ALTER TABLE with the TYPE option of RAW or STATIC. For example, the following statement changes table **tablog** to a RAW table:

```
ALTER TABLE tablog TYPE (RAW)
```

Altering a Table to Turn On Logging

To switch from a nonlogging table to a logging table, use the SQL statement ALTER TABLE with the TYPE option of STANDARD or OPERATIONAL. For example, the following statement changes table **tabnolog** to a STANDARD table:

```
ALTER TABLE tabnolog TYPE (STANDARD)
```

Warning: *When you alter a table to STANDARD or OPERATIONAL from any other table type, you turn logging on for that table. After you alter the table, perform a level-0 backup if you need to be able to restore the table.*

Creating a Nonlogging Temporary Table

When you create a temporary table with the SELECT...INTO TEMP statement, the temporary table is a logging table by default. If you do not care to log transactions in this temporary table, specify the WITH NO LOG clause on the SELECT...INTO TEMP statement or create a scratch table.

The database server uses the dbspaces specified in the DBSPACETEMP configuration parameter or DBSPACETEMP environment variable when you specify either of the following items:

- SELECT...INTO TEMP <*temp_table*> WITH NO LOG
- SELECT...INTO SCRATCH <*temp_table*>

If you use the default value of NONCRITICAL for DBSPACETEMP, the optimizer stores temporary tables in any dbspace that does not contain critical files such as logical logs.

For more information on improving performance with temporary tables, see your *Performance Guide* and "Temporary Tables" on page 15-30.

Monitoring Transaction Logging

This section discusses ways to monitor the logging status of your database and tables. For information on monitoring I/O and memory usage for logging and checkpoints, see your *Performance Guide*.

Monitoring Transaction Logging with SMI Tables

Query the **sysdatabases** table in the **sysmaster** database to determine the logging status. This table contains a row for each database that the database server manages. The **flags** field indicates the logging status of the database. The **is_logging**, **is_buff_log**, and **is_ansi** fields indicate whether logging is active, and whether buffered logging or ANSI-compliant logging is used. For a description of the columns in this table, see the **sysdatabases** section in the chapter about the **sysmaster** database in the *Administrator's Reference*.

Monitoring Transaction Logging with System Catalog Tables

Query the **systables** system catalog to determine the table type. If the flag value is `0002` (raw) or `0004` (static), the table is not logged. For more information, see the *Informix Guide to SQL: Reference*.

Chapter

Logical Log

20

In This Chapter

As the database server administrator, you have responsibilities to configure and manage the logical log. These responsibilities include the following tasks:

- Learning about the logical log
- Allocating an appropriate amount of disk space for the logical log
- Choosing an appropriate location for the logical-log files
- Monitoring the logical-log file status
- Backing up the logical-log files to media

The information in this chapter will help you understand these tasks and the nature of the logging process. For information on how to perform other logical-log tasks, see Chapter 21, "Managing Logical-Log Files."

Logical Log

To keep a history of database and database server changes since the time of the last storage-space backup, the database server generates and stores log records. The database server stores the log records in the *logical log*, which is made up of logical-log files. The log is called *logical* because the log records represent units of work related to the logical operations of the database server, as opposed to physical operations. At any time, the combination of a storage-space backup plus logical-log backup contains a complete copy of your database server data.

Logical-Log Files

Logical-log files are not files in the operating-system sense of the word *file*. Each logical-log file is a separate allocation of disk space that the database server manages. You must always have at least three logical-log files in the logical log.

The database server administrator needs to be concerned with the logical-log files that make up the logical log. If the files are not managed properly, the database server can suspend processing and, in the worst case, shut down.

The database server administrator must choose an appropriate number, size, and physical location for logical-log files. The following sections discuss these topics:

- "Size of the Logical Log" on page 20-6
- "Location of Logical-Log Files" on page 20-9

The database server administrator must also ensure that the next logical-log file is always backed up and free. The following sections discuss this topic:

- "Identification of Logical-Log Files" on page 20-9
- "Backup of Logical-Log Files" on page 20-12
- "Freeing of Logical-Log Files" on page 20-13

Most database users might be concerned with whether transaction logging is buffered or whether a table uses logging.

Logical-Log Administration

All the databases managed by a single database server instance store their log records in the same logical log, regardless of whether they use transaction logging or whether their transaction logging is buffered. For information on transaction logging, see Chapter 18, "Logging." If you want to change the database-logging status, see "Settings or Changes for Logging Status or Mode" on page 18-12.

Most end users should not be concerned with the logical-log files. The primary administrative tasks include managing individual logical-log files and determining how much disk space to allocate to the logical log.

Logical-Log Files on a Coserver

You can also create individual logical-log files on a specified coserver or a logslice across many coservers. For example, you can vary the number of logical-log files per coserver in a way that reflects the different requirements of each coserver. Each coserver must have a minimum of three logical-log files.

For information on the **onutil** CREATE COGROUP, **onutil** CREATE LOGSLICE, and **onutil** ALTER LOGSLICE commands, see the utilities chapter in the *Administrator's Reference*.

Logslices

A *logslice* is a set of logical-log files that occupy a dbslice and are owned by multiple coservers, one logical-log file per dbspace. Logslices simplify the process of adding and deleting logical-log files by treating sets of them as single entities. You cannot perform operations on the individual logical-log files that are part of a logslice.

A *dbslice* is a named set of dbspaces that you can manage as a single storage object. If a dbslice has multiple dbspaces per coserver, each coserver has multiple logical-log files. You can add multiple logslices to a dbslice as long as all the dbspaces have enough free space to accommodate the new logical-log files. Each logslice is a distinct set of logical-log files. Logical-log files are not shared across logslices.

After you add dbspaces to a dbslice, you can add logical-log files to a logslice.

Size of the Logical Log

In determining how much disk space to allocate, you must balance disk space and performance considerations. If you allocate more disk space than necessary, space is wasted. If you do not allocate enough disk space, however, performance might be adversely affected.

Performance Considerations

For a given level of system activity, the less logical-log disk space that you allocate, the sooner that logical-log space fills up, and the greater the likelihood that user activity is blocked due to logical-log backups and checkpoints, as follows:

- Logical-log backups

  When the logical-log files that make up the logical log fill, you have to back them up. The backup process can hinder transaction processing that involves data located on the same disk as the logical-log files. If enough logical-log disk space is available, however, you can wait for periods of low user activity before you back up the logical-log files. (See "Backup of Logical-Log Files" on page 20-12.)

- Checkpoints

  At least one checkpoint record must always be written to the logical log. If you need to free the logical-log file that contains the last checkpoint, the database server must write a new checkpoint record to the current logical-log file. If the frequency with which logical-log files are backed up and freed increases, the frequency at which checkpoints occur increases. Although checkpoints block user processing, they no longer last as long. Because other factors (such as the physical-log size) also determine the checkpoint frequency, this effect might not be significant. (See "Freeing of Logical-Log Files" on page 20-13.)

- Table logging

  Whether tables use logging also affects the rate at which the logical log fills.

These performance considerations are related to how fast the logical log fills. The rate at which the logical log fills, in turn, depends on other factors such as the level of user activity on your system. You need to tune the logical-log size, therefore, to find the optimum value for your system.

Long-Transaction Considerations

In addition to the performance considerations discussed in the previous section, you risk a long-transaction situation if logical-log disk space is insufficient. For more information on the long-transaction situation, refer to "Logical Log and Long Transactions" on page 20-15.

Size and Number of Logical-Log Files

After you know how much disk space to allocate for the entire logical log, you can make decisions about how many log files you want and what size.

When you think about the size of the logical-log files, consider these points:

- The minimum size for a logical-log file is 200 kilobytes.
- The maximum size for a logical-log file is essentially unbounded.
- If your tape device is slow, ensure that logical-log files are small enough to be backed up quickly.
- Smaller log files mean slower recovery because you potentially lose the last unbacked-up logical-log file if the disk that contains the logical-log files goes down.

Size of the Logical Log

Use the LOGSIZE configuration parameter to set the size of the logical log. It is difficult to predict how much logical-log space your database server system requires until it is fully in use. The following expression provides the *minimum* total-log-space configuration, in kilobytes, that Informix recommends:

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

Set *users* to the maximum number of users that you expect to access the database server concurrently. If you set the NETTYPE parameter, you can use the value that you assigned to the NETTYPE **users** field. If you configured more than one connection by setting multiple NETTYPE configuration parameters in your configuration file, sum the **users** fields for each NETTYPE, and substitute this total for *users* in the preceding formula.

Set *maxrows* to the maximum number of rows that you expect in the tables.

You can increase the amount of space devoted to the logical log as necessary and in several ways. The easiest way is to add another logical-log file. See "Adding a Logical-Log File or Logslice" on page 21-4.

Number of Logical-Log Files

When you think about the number of logical-log files, consider these points:

- You must always have at least three logical-log files.
- You should create enough logical-log files so that you can switch log files if needed without running out of free logical-log files.
- The number of logical-log files cannot exceed the value of the ONCONFIG parameter LOGSMAX.
- The number of logical-log files affects the frequency of logical-log backups.

The LOGFILES parameter provides the number of logical-log files. The LOGSIZE parameter provides the size of the logical-log files that are created when the database server initializes disk space. The database server administrator sets both of these configuration parameters in the ONCONFIG file. If all your logical-log files are the same size, you can calculate the total space allocated to the logical-log files as follows:

```
total logical log space = LOGFILES * LOGSIZE
```

If you add logical-log files that are not the size specified by LOGSIZE, you cannot use the (LOGFILES * LOGSIZE) expression to calculate the size of the logical log. Instead, you need to add the sizes for each individual log file on disk. For information on how to access the size of logical-log files, see "Monitoring the Logical Log for Fullness" on page 21-16.

For information on LOGSIZE, LOGFILES, and NETTYPE, see the chapter on configuration parameters in the *Administrator's Reference*.

Location of Logical-Log Files

When the database server initializes disk space, it places the logical-log files and the physical log in the root dbspace. You have no control over this action. To improve performance (specifically, to reduce the number of writes to the root dbspace and minimize contention), move the logical-log files out of the root dbspace to a dbspace on a disk that is not shared by active tables or the physical log. See "Moving a Logical-Log File to Another Dbspace" on page 21-7.

To improve performance further, separate the logical-log files into two groups and store them on two separate disks (neither of which contains data). For example, if you have six logical-log files, you might locate files 1, 3, and 5 on disk 1, and files 2, 4, and 6 on disk 2. This arrangement improves performance because the same disk drive never has to handle writes to the current logical-log file and backups to tape at the same time.

The logical-log files contain critical information and should be mirrored for maximum data protection. If you move logical-log files to a different dbspace, plan to start mirroring on that dbspace.

Identification of Logical-Log Files

Each logical-log file, whether backed up to media or not, has a *unique ID* number. The sequence begins with 1 for the first logical-log file filled after you initialize the database server disk space. When the current logical-log file becomes full, the database server switches to the next logical-log file and increments the unique ID number for the new log file by one.

The actual disk space allocated for each logical-log file has an identification number known as the *logid*. For example, if you configure six logical-log files, these files have logid numbers one through six. As logical-log files are backed up and freed, the database server reuses the disk space for the logical-log files. However, the database server continues to increment the unique ID numbers by one. Figure 20-1 on page 20-10 illustrates the relationship between the logid numbers and the unique ID numbers.

Figure 20-1
Logical-Log File-Numbering Sequence

Logid Number	First Rotation Unique ID Number	Second Rotation Unique ID Number	Third Rotation Unique ID Number	Fourth Rotation Unique ID Number
1	1	7	13	19
2	2	8	14	20
3	3	9	15	21
4	4	10	16	22
5	5	11	17	23
6	6	12	18	24

For information on how to display the unique ID and logid numbers of a logical-log file, refer to "Monitoring the Logical Log for Fullness" on page 21-16.

Status Flags of Logical-Log Files

All logical-log files have one of the following three status flags in the first position: Added (A), Free (F), or Used (U). Descriptions of all the individual logical-log status flags follow.

Status Flag	Description
Added (A)	A logical-log file has an *added* status when it is newly added. The logical-log file does not become available for use until you complete a level-0 backup of the root dbspace.
Free (F)	A logical-log file is *free* when it is available for use. A logical-log file is freed after it is backed up, all transactions within the logical-log file are closed, and the oldest update stored in this file is flushed to disk.
Used (U)	A logical-log file is *used* when it is still needed by the database server for recovery (rollback of a transaction or finding the last checkpoint record).
Backed-Up (B)	A logical-log file has a *backed-up* status after it has been backed up.
Current (C)	A logical-log file has a *current* status if the database server is currently filling the log file.
Last (L)	A logical-log file has a status of *last* if it contains the most recent checkpoint record in the logical log. This file and subsequent files cannot be freed until the database server writes a new checkpoint record to a different logical-log file.

Figure 20-2 shows the possible log-status flag combinations.

Figure 20-2
Logical-Log Status Flags

Status Flag	Status of Logical-Log File
A------	Log has been added since the last level-0 storage-space backup. Not available for use.
F------	Log is free. Available for use.
U	Log has been used but not backed up.

(1 of 2)

Status Flag	Status of Logical-Log File
U-B----	Log is backed up but still needed for recovery.
U-B---L	Log is backed up but still needed for recovery. Contains the last checkpoint record.
U---C	Log is the current logical-log file.
U---C-L	Log is the current logical-log file. It contains the last checkpoint record.

(2 of 2)

Tip: *A logical-log file has a status flag of* F *only if the system has been reinitialized.*

To find out the status of a logical-log file, use the methods explained in "Monitoring the Logical Log for Fullness" on page 21-16.

Backup of Logical-Log Files

The logical logs contain a history of the transactions that have been performed. The process of copying a logical-log file to media is referred to as *backing up* a logical-log file. Backing up logical-log files achieves the following two objectives:

- It stores the logical-log records on media so that they can be rolled forward if a data restore is needed.
- It makes logical-log-file space available for new logical-log records.

You can initiate a manual logical-log backup or set up continuous logical-log backups. If you use ON-Bar to perform logical-log backups, see the *Backup and Restore Guide*.

Logical-Log Restore

After you restore the storage spaces, you must restore the logical logs to bring all the data to a consistent state. A restore performed when the database server is on-line is called a warm restore.

The database server automatically skips logical replay during a warm restore when the dbspaces have not participated in a transaction. In the **onstat -d** output, the **S** flag displays for a dbspace that is a candidate for skipping logical replay. For information on logical replay, see the *Backup and Restore Guide*.

Point-In-Time Restore

You can restore storage spaces and logical logs to a particular point in time during a cold restore. If you use ON-Bar to perform point-in-time restores, see the *Backup and Restore Guide*.

Freeing of Logical-Log Files

If you back up a logical-log file, that file is not necessarily free to receive new log records. The following criteria must be satisfied before the database server frees a logical-log file for reuse:

- The log file is backed up.
- No records within the logical-log file are associated with open transactions.
- The logical-log file does not contain the oldest update not yet flushed to disk. Perform a full checkpoint using the **onmode -c** command to ensure that the logical-log file does not contain the oldest update.

Tip: *You can free a logical log with a status of* U-B *(and not* L*) only if it is not spanned by an active transaction and does not contain the oldest update.*

Database Server Attempt to Free a Log File

The database server attempts to free logical-log files each time that the database server commits or rolls back a transaction, it attempts to free the logical-log file in which the transaction began.

The attempt succeeds only if the criteria listed in the preceding section, "Freeing of Logical-Log Files," are met.

Action If the Next Logical-Log File Is Not Free

If the database server attempts to switch to the next logical-log file but finds that the next log file in sequence is still in use, the database server immediately suspends all processing. Even if other logical-log files are free, the database server cannot skip a file in use and write to a free file out of sequence. Processing stops to protect the data within the logical-log file.

The logical-log file might be in use for any of the following reasons:

- The file contains the latest checkpoint or the oldest update not yet flushed to disk.

  Issue the **onmode -c** command to perform a full checkpoint and free the logical-log file. For more information, see "Forcing a Full Checkpoint" on page 24-8.

- The file contains an open transaction.

  The open transaction is the long transaction discussed in "Logical Log and Long Transactions" on page 20-15. In this situation, you have to recover the database server data from storage-space backups in a full-system restore.

- The file is not backed up.

  If the logical-log file is not backed up, processing resumes when you perform the backup. Use ON-Bar to back up the logical-log files.

The database server does not suspend processing when the next log file contains the last checkpoint or the oldest update. The database server always forces a full checkpoint when it switches to the last available log, if the previous checkpoint record or oldest updated not yet flushed to disk is located in the log that follows the last available log. For example, if four logical-log files have the status shown in the following list, the database server forces a checkpoint when it switches to logical-log file 3.

logid	Logical-Log File Status
1	U-B----
2	U---C--
3	F
4	U-B---L

Logical Log and Long Transactions

A *long transaction* is a transaction that starts in one logical-log file and is not committed when the database server needs to reuse that same logical-log file. In other words, a long transaction spans more than the total space allocated to the logical log.

Because the database server cannot free a logical-log file until all records within the file are associated with closed transactions, the long transaction prevents the first logical-log file from becoming free and available for reuse.

To prevent long transactions from developing, take the following precautions:

- Ensure that the logical-log file does not fill too fast.
- Ensure that transactions do not remain open too long.
- Set high-water marks to have the database server automatically slow down processing when a long transaction is developing.

The subsequent sections explain these steps.

Factors That Influence the Rate at Which Logical-Log Files Fill

Several factors influence how fast the logical log fills. It is difficult to know exactly which factor is the most important for a given instance of the database server, so you need to use your own judgment to estimate how quickly your logical log fills and how to prevent long-transaction conditions. Consider these factors:

- Size of the logical log

  A smaller logical log fills faster than a larger logical log. If you need to make the logical log larger, you can add another logical-log file, as explained in "Adding a Logical-Log File or Logslice" on page 21-4.

- Number of logical-log records

  The more logical-log records written to the logical log, the faster it fills. If databases that your database server manages use transaction logging, transactions against those databases fill the logical log faster than transactions against databases without transaction logging.

  When you use logging tables (STANDARD or OPERATIONAL), the logical log fills faster than when you use nonlogging tables. For more information, refer to "Logging and Nonlogging Tables" on page 18-8.

- Type of log buffering

  As explained in "Unbuffered Transaction Logging" on page 18-10, databases that use unbuffered transaction logging fill the logical log faster than databases that use buffered transaction logging.

- Size of individual logical-log records

  The sizes of the logical-log records vary, depending on both the processing operation and the database server environment. In general, the longer the data rows, the larger the logical-log records. The logical log contains images of rows that have been inserted, updated, or deleted. Also, updates can use up to twice as much space as inserts and deletes because they might contain both before-images and after-images. Inserts store only the after-image and deletes store only the before-image.

- Frequency of rollbacks

  The frequency of rollbacks affects the rate at which the logical log fills. More rollbacks fill the logical log faster. The rollbacks themselves require logical-log file space although the rollback records are small. In addition, rollbacks increase the activity in the logical log.

Factors That Prevent Closure of Transactions

Several factors influence when transactions close. Be aware of these factors so that you can prevent long-transaction problems:

- Transaction duration

  The duration of a transaction might be beyond your control. For example, a client that does not write many logical-log records might cause a long transaction if the users permit transactions to remain open for long periods of time. (For example, a user who is running an interactive application might leave a terminal to go to lunch part of the way through a transaction.)

  The larger the logical-log space, the longer a transaction can remain open without a long-transaction condition developing. However, a large logical log by itself does not ensure that long transactions do not develop. Application designers should consider the transaction-duration issue, and users should be aware that leaving transactions open can be detrimental.
- High CPU and logical-log activity

  The amount of CPU activity can affect the ability of the database server to complete the transaction. Repeated writes to the logical-log file increase the amount of CPU time that the database server needs to complete the transaction. Increased logical-log activity can imply increased contention of logical-log locks and latches as well.

Setting High-Water Marks

The database server alters processing at two critical points to manage the long-transaction condition. To tune both points, you can set values in the ONCONFIG file.

The first critical point is the *long-transaction high-water mark*. When the logical log reaches the long-transaction high-water mark, the database server recognizes that a long transaction exists and begins searching for an open transaction in the oldest, used (but not freed) logical-log file. If a long transaction is found, the database server directs the thread to begin to roll back the transaction. More than one transaction can be rolled back if more than one long transaction exists.

The transaction rollback itself generates logical-log records, however, and as other processes continue writing to the logical-log file, the logical log continues to fill.

The second critical point is the *exclusive-access, long-transaction high-water mark*. When the logical log reaches the exclusive-access, long-transaction high-water mark, the database server dramatically reduces log-record generation. Most threads are denied access to the logical log. Only threads that are currently rolling back transactions (including the long transaction) and threads that are currently writing COMMIT records are allowed access to the logical log. Restricting access to the logical log preserves as much space as possible for rollback records that are being written by the user threads that are rolling back transactions.

If the long transactions cannot be rolled back before the logical log fills, the database server shuts down. If this situation occurs, you must perform a data restore. During the data restore, you must *not* roll forward the last logical-log file. Doing so re-creates the problem by filling the logical log again.

The default values for the configuration parameters LTXHWM and LTXEHWM are `50` and `60`, respectively. These values eliminate any risk of a long transaction having too little log space in which to roll back. The database server initialization emits a warning if your ONCONFIG file contains values greater than `50` and `60` for these parameters. To overcome these warnings, reduce your parameters to conform. If your log space is finely tuned such that your LTXHWM percentage represents precisely what your longest transaction requires, you will need to add an amount to your log space equal to the difference between your current LTXHWM value and the recommended value of `50`.

For information on LTXHWM and LTXEHWM, see the chapter on configuration parameters in the *Administrator's Reference*.

Logs-Full High-Water Mark

To enable the logs-full high-water mark, set the LBU_PRESERVE configuration parameter to 1. When you set LBU_PRESERVE to 1, the database server blocks DB-Access, ESQL/C, and all other clients from generating log records in the last logical-log file when the logs-full condition is reached. The default value of LBU_PRESERVE is 0, or off.

Whenever you change the value of LBU_PRESERVE, you must reinitialize shared memory for the change to take effect.

The LBU_PRESERVE configuration parameter, when set to 1, enables the logical-log high-water mark. The high-water mark prevents transaction records from filling the last free logical-log file. When the high-water mark has passed, client transaction requests are frozen until the logical log is backed up.

Extended Parallel Server does not use emergency log backup which is discussed in the next section. Instead, it automatically sets the LBU_PRESERVE parameter to ensure that the last free log is reserved for backup and restore operations. You cannot change the LBU_PRESERVE value in the ONCONFIG file. If LOG_BACKUP_MODE is set to MANUAL or CONT, LBU_PRESERVE is enabled for all coservers. If LOG_BACKUP_MODE is set to NONE, LBU_PRESERVE is disabled for all coservers.

Emergency Log Backup

Although the logs-full high-water mark eliminates the need for emergency backup during transaction processing, four known scenarios still require the database server administrators to use emergency logical-log backup. Each case is examined in detail in the following sections.

System-Monitoring Interface

A privileged client is responsible for building the system-monitoring interface (SMI). This client can potentially invade the last logical-log file. If you do not configure sufficient log space or a sufficient number of logical-log files, the privileged client might not succeed in building SMI without a logical-log backup. This situation can cause the logical log to fill.

Fast Recovery

When you start the database server after an uncontrolled shutdown, it needs log space to roll back any transactions that were uncommitted when the shutdown occurred. The threads that perform the recovery have privileges that allow them to use the last logical-log file. Because of this privilege, the logical log might become full, but only in the unlikely case that the number and size of transactions open when the shutdown occurred exceed the size of the logical log.

Small Logs, Many Users

Perhaps you configure your logical log files as follows:

```
Logical Log Size < 2 * page_size * number of users
```

If all users enter transactions of maximum complexity, applications might invade the last logical-log file with OLTP activity. Only when you set the size of the logical log much smaller than two pages per user can a logs-full condition occur.

Administrative Activity When Logs Need Backing Up

Because certain administrative utilities have the privilege to invade the last logical-log file, you might have to perform an emergency logical-log backup. For example, when the logical log approaches full while you are doing large quantities of administrative work, you might need to perform an emergency logical-log backup.

Extended Parallel Server uses the following utilities.

onbar_d	onbar_m	onbar_w
ondblog	onsmsync	onutil

Logging Process

This section describes in detail the logging process for dbspaces. This information is not required for performing normal database server administration tasks.

Dbspace Logging

The database server uses the following logging process for operations that involve data stored in dbspaces:

1. Read the data page from disk to the shared-memory page buffer.
2. Copy the unchanged page to the physical-log buffer, if needed.
3. Write the new data to the page buffer, and create a logical-log record of the transaction, if needed.
4. Flush the physical-log buffer to the physical log on disk.
5. Flush the logical-log buffer to a logical-log file on disk.
6. Flush the page buffer, and write it back to disk.

Read Page into Shared-Memory Buffer Pool

In general, an insert or an update begins when a thread requests a row. The database server identifies the page on which the row resides and attempts to locate the page in the shared-memory buffer pool. If the page is not already in shared memory, the database server reads the page from disk. "Database Server Thread Access to Buffer Pages" on page 13-42 explains this process in more detail.

Copy the Page Buffer to the Physical-Log Buffer

Before the database server modifies a dbspace data page for a nonfuzzy operation, it stores a copy of the unchanged page in the physical-log page buffer if it is needed for fast recovery. The database server eventually flushes the physical-log page buffer that contains this *before-image* to the physical log on disk. Until the database server performs a new checkpoint, subsequent modifications to the same page do not require another before-image to be stored in the physical-log buffer.

The database server knows if a page is already in the physical log. If the time stamp on the page is more recent than the time stamp for the last checkpoint, the page has been changed since the checkpoint and is therefore already in the physical log.

For fuzzy operations, the database server does not store a copy of the before-image of the page into the physical-log page buffer. For fast recovery, the database server finds the oldest update in the logical log. For more information, refer to "Details of Fast Recovery After A Fuzzy Checkpoint" on page 24-18.

Read Data into Buffer and Create Logical-Log Record

The thread that performs the modifications receives data from the application. After the database server stores a copy of the unchanged data page in the physical-log buffer, the thread writes the new data to the page buffer and writes records necessary to roll back or re-create the operation to the logical-log buffer. For more information, refer to "When the Logical-Log Buffer Becomes Full" on page 13-53.

Flush Physical-Log Buffer to the Physical Log

The database server must flush the physical-log buffer before it flushes the data buffer. Flushing the physical-log buffer ensures that a copy of the unchanged page is available until the changed page is written to the physical log. For more information, refer to "Flushing the Physical-Log Buffer" on page 13-46.

Flush Page Buffer

After the database server flushes the physical-log buffer, the database server flushes the data buffer and writes the modified data pages for non-fuzzy operations to disk at the next fuzzy checkpoint. The database server writes all the modified data pages to disk at the next full checkpoint, or when a page cleaner determines that the page should be written to disk. The database server does not flush the data buffer as the transaction is committed. For more information, see "Flushing Data to Disk" on page 13-45.

Flush Logical-Log Buffer

To flush the logical-log buffer, the database server writes the logical-log records to the current logical-log file on disk. For more information, see "Flushing the Logical-Log Buffer" on page 13-52.

For information on the criteria that must be satisfied before the database server frees a logical-log file for reuse, see "Freeing of Logical-Log Files" on page 20-13.

Chapter

Managing Logical-Log Files

21

In This Chapter

This chapter contains information on managing the database server logical-log files with command line utilities. You must manage logical-log files even if none of your databases uses transaction logging.

The chapter covers the following tasks:

- Adding a logical-log file
- Dropping a logical-log file
- Moving a logical-log file
- Changing the size of a logical-log file
- Changing the logical-log configuration parameters
- Freeing a logical-log file
- Switching to the next logical-log file
- Monitoring log-backup status
- Creating, altering, and dropping a logslice

For background information regarding the logical log, refer to Chapter 20, "Logical Log." When you add or drop logical-log files, the database server must be in quiescent mode.

When you add or drop logslices, the database server must be in quiescent mode.

You must log in as either **informix** or **root** on UNIX to make any of the changes described in this chapter.

Backing Up Logical-Log Files

After you add, move, or delete a logical-log file, you must perform a level-0 backup of the root dbspace and the modified dbspace. For instructions on backing up logical-log files with ON-Bar, refer to the *Backup and Restore Guide*.

Adding a Logical-Log File or Logslice

You might add a logical-log file or logslice for the following reasons:

- To increase the disk space allocated to the logical log
- To change the size of your logical-log files or logslice
- As part of moving logical files or logslices to a different dbspace

Verify that you do not exceed the maximum number of logical-log files and logslices allowed in your configuration, specified as LOGSMAX.

If you need to, you can increase LOGSMAX. See "Changing LOGSMAX, LTXHWM, or LTXEHWM in the ONCONFIG File" on page 21-12. Also check the values of LOGFILES and LOGSIZE and increase them if necessary. See "Using a Text Editor to Change LOGSIZE or LOGFILES" on page 21-11.

Adding a Logical-Log File or Logslice with onutil

Use the **onutil** utility to add a logical-log file or logslice. You can use either the default log-file size or specify a new size.

You add logical-log files or logslices one at a time with the database server in quiescent mode. You cannot add a logical-log file or logslice during a storage-space backup.

Adding a Logical-Log File

For an example of using the **onutil** CREATE LOG command, see the utilities chapter in the *Administrator's Reference*. You can specify the dbspace or coserver where the logical-log file resides. Adding a log file of a new size does not change the value of LOGSIZE.

The status of the new log file is A. The newly added log file becomes available after you create a level-0 backup of the root dbspace and the dbspaces that contain the log file on all coservers.

Adding a Logslice

For an example of using the **onutil** CREATE LOGICAL LOGSLICE command to add a logslice in a dbslice, see the utilities chapter in the *Administrator's Reference*. You can specify the dbslice where the logslice resides.

Altering a Logslice

After you add dbspaces to a dbslice in a logslice, you can use the **onutil** ALTER LOGSLICE ADD LOGS command to add logical-log files in the new dbspaces. Execute the following command to add log files to logslice **mylogslice**. Each new dbspace in the dbslice gets a new logical-log file.

```
% onutil
ALTER LOGSLICE mylogslice ADD LOGS;
```

After you alter a logslice, perform a level-0 backup of the root dbspace and the dbspaces that contain the logslice on all coservers to enable access to the new logical-log files. For more information on **onutil** ALTER LOGSLICE ADD LOGS, see the utilities chapter of the *Administrator's Reference*.

Dropping a Logical-Log File or Logslice

You can drop a logical-log file or logslice to increase the amount of the disk space available within a dbspace. The database server requires a minimum of three logical-log files per coserver (three logslices) at all times. You can also drop a logslice to increase the amount of the disk space available within a dbslice. For more information, see "Dropping a Logical-Log File or Logslice with onutil" on page 21-6.

To make this change, you must log in as either **informix** or **root**, and the database server must be in quiescent mode.

Log files or logslices that are newly added and have status A do not count toward this minimum of three. You cannot drop a log if your logical log is composed of only three log files.

You drop log files or logslices one at a time. You can only drop a log file or logslice that has a status of Free (F) or newly Added (A). You must know the logid number of each logical log or logslice that you intend to drop.

For information on obtaining a display of the logical-log files and logid numbers, see "Displaying Logical-Log Records" on page 21-19.

Dropping a Logical-Log File or Logslice with onutil

Use **onutil** to drop a logical-log file or logslice. To obtain the logid, use the **xctl onstat -l** command. You might want to back up the old logical-log files and logslices before you drop them.

After you drop the log file or logslice, create a level-0 backup of the root dbspace and dbspaces that contain the log files on all coservers. This action ensures that the backup copy of the reserved pages contains information about the current number of logical-log files or logslices. This information prevents the database server from attempting to use the dropped log files or logslices during a restore. For information on creating a level-0 backup with ON-Bar, refer to the *Backup and Restore Guide*.

Dropping a Log File

Execute the following command to drop a logical-log file, whose logid number is 6, that is on coserver **eds.2**:

```
onutil
1> DROP LOG 6 COSERVER eds.2;
```

For information on the **onutil** DROP LOGICAL LOG command, see the utilities chapter in the *Administrator's Reference*.

Dropping a Logslice

You drop logslices one at a time. You can only drop a logslice that has a status of Free (F) or newly Added (A). You must know the logid number of each logslice that you intend to drop.

Execute the following command to drop a logslice whose name is **logslice2**:

```
onutil
1> DROP LOGSLICE logslice2;
```

For information on the **onutil** DROP LOGSLICE command, see the utilities chapter in the *Administrator's Reference*.

Moving a Logical-Log File to Another Dbspace

You might want to move a logical-log file for performance reasons or to make more space in the dbspace, as explained in "Location of Logical-Log Files" on page 20-9. To find out the location of logical-log files, see "Monitoring Logging Activity" on page 21-15.

Changing the location of the logical-log files is actually a combination of two simpler actions:

- Dropping logical-log files from their current dbspace
- Adding the logical-log files to their new dbspace

The database server must be in quiescent mode to move a logical-log file. Although moving the logical-log files is not difficult, it can be time-consuming.

The following procedure provides an example of how to move six logical-log files from the **root** dbspace to another dbspace, **dbspace_1**.

To move the logical-log files

1. Free all log files except the current log file.

   See "Freeing a Logical-Log File" on page 21-13.

2. Verify that the value of LOGSMAX is greater than or equal to the number of log files after the move plus 3.

   In this case, the value of LOGSMAX must be greater than or equal to 9. Change the value of LOGSMAX, if necessary. See "Changing LOGSMAX, LTXHWM, or LTXEHWM in the ONCONFIG File" on page 21-12.

3. Drop all but three of the logical-log files.

   You cannot drop the current logical-log file. If you have only three logical-log files in the root dbspace, skip this step.

   See "Dropping a Logical-Log File or Logslice" on page 21-5.

4. Add the new logical-log files to the different dbspace. In this case, add six new logical-log files to **dbspace_1**.

   See "Adding a Logical-Log File or Logslice" on page 21-4.

5. Create a level-0 backup of the root dbspace and the dbspaces that contain the log files to make the new logical-log files available to the database server. For more information, see "Backing Up Logical-Log Files" on page 21-4.

6. Switch the logical-log files to start a new current log file.

   See "Switching to the Next Logical-Log File" on page 21-15.

7. Back up the former *current logical-log file* to free it.

8. Drop the three logical-log files that remain in the root dbspace.

Changing the Size of Logical-Log Files

You can change the size of logical-log files or logslices in the following ways:

- Use a text editor to change the LOGSIZE or LOGFILES parameter in the ONCONFIG file.
- Use **onutil** to drop the old logical-log file or logslice and to create a new one of a different size.

  This change has no effect on the LOGSIZE parameter. See "Using onutil to Change the Size of a Log File or Logslice" on page 21-9.

Using a Text Editor to Change the Size of a Log File

Changing LOGSIZE changes the default size for all subsequent logical-log files and logslices added but is time-consuming because it requires that you reinitialize the database server to see the change. Subsequent log files are the new size. See "Using a Text Editor to Change LOGSIZE or LOGFILES" on page 21-11.

Using onutil to Change the Size of a Log File or Logslice

You can use **onutil** to add a new logical-log file or logslice with a different size than LOGSIZE.

To use onutil to change the size of a logical-log file

1. Ensure that the database server is in quiescent mode.

   ```
   xctl onmode -sy
   ```
2. Change the value of LOGSIZE in the ONCONFIG file.
3. If the database server contains the maximum number of logical logs (specified in LOGSMAX), use the **onutil** DROP LOG command to drop the old logical log before you add the new logical log.

4. Use the **onutil** CREATE LOG command to add a logical-log file, specifying the new size. The following example shows how to add three larger logical-log files to each dbspace on **coserver eds.3**:

   ```
   % onutil
   1> CREATE LOG dbspace logspace3
   2> SIZE 50 MBYTES;
   Logical log successfully added.
   4> CREATE LOG dbspace logspace4
   5> SIZE 50 MBYTES;
   Logical log successfully added.
   7> CREATE LOG dbspace logspace5
   8> SIZE 50 MBYTES;
   Logical log successfully added.
   ```

5. Back up the root dbspace and the dbspaces that contain the logs on all coservers to enable your new logical logs.
6. Execute a series of **onmode -l** commands to determine which old log is current and to make one of the new logs the current log.

For more information on the **onutil** DROP LOG command, **onutil** CREATE LOGICAL LOG command, **onutil** DROP LOGSLICE command, and **onutil** CREATE LOGSLICE command, see the utilities chapter in the *Administrator's Reference*.

To use onutil to change the size of a logslice

1. Ensure that the database server is in quiescent mode.

   ```
   xctl onmode -sy
   ```

2. Use the **onutil** DROP LOGSLICE command to drop the old logslice. For an example, see "Dropping a Logslice" on page 21-7.
3. Use the **onutil** CREATE LOGSLICE command to add a logslice, specifying the new size. The following example adds a 50-megabyte logslice to coserver **eds.3**:

   ```
   % onutil
   1> CREATE LOGSLICE mylogslice
   2> SIZE 50 MBYTES;
   Logslice successfully added.
   ```

4. Back up the root dbspace and the dbspaces that contain the logslices on each coserver to enable your new logslices.

Changing Logical-Log Configuration Parameters

The following configuration parameters affect the logical-log file and how the database server works with it:

- LOGSIZE
- LOGFILES
- LOGSMAX
- LTXHWM
- LTXEHWM

The following sections explain the procedure for changing each of these configuration parameters. For more information on these parameters, see the chapter on configuration parameters in the *Administrator's Reference*.

You can use a text editor to change these parameters in the ONCONFIG file. You must be logged in as **root** or **informix** to change these configuration parameters.

Using a Text Editor to Change LOGSIZE or LOGFILES

If you want to change the size of the log files, you might find it easier to add new log files of the desired size and then drop the old ones.

Important: *The changes to LOGSIZE and LOGFILES do not take effect until you reinitialize the disk space.*

To change the size or number of logical-log files and logslices

1. Bring the database server off-line or into quiescent mode.
2. To change the size of the log files, change the value of LOGSIZE in the ONCONFIG file.
3. To change the number of the log files, change the value of LOGFILES in the ONCONFIG file. You might also need to increase the LOGSMAX value.

4. Unload all the database server data.

   To retain your existing data when you reinitialize the disk, you must unload the data beforehand and reload it once the disk is initialized. This process makes changing these parameters relatively difficult. You cannot rely on storage-space backups to unload and restore the data because a restore returns the parameters to their previous value.

5. Reinitialize disk space.

   After the database server disk space is reinitialized, re-create all databases and tables. Then reload all database server data. For more information, see "Initializing Disk Space" on page 9-5.

6. Re-create all databases and tables.
7. Reload all the database server data.

   For information on loading and unloading data, see the *Informix Migration Guide*.

8. Back up the root dbspace to enable your changed logical logs or logslices.

Changing LOGSMAX, LTXHWM, or LTXEHWM in the ONCONFIG File

You can use a text editor to change the value of LOGSMAX, LTXHWM, or LTXEHWM while the database server is on-line. You must be logged in as **root** or **informix** on UNIX to change these configuration parameters.

Changes to these configuration parameters take effect when you shut down and restart the database server. For more information on LOGSMAX, LTXHWM, or LTXEHWM, see the chapter on configuration parameters in the *Administrator's Reference*.

Freeing a Logical-Log File

For a description of what constitutes a free logical-log file, see “Status Flags of Logical-Log Files” on page 20-11.

You might want to free a logical-log file for the following reasons:

- So that the database server does not stop processing
- To free the space used by deleted blobpages

The procedures for freeing log files vary, depending on the status of the log file. Each procedure is described in the following sections. To find out the status of logical-log files, see “Monitoring Logging Activity” on page 21-15.

Tip: *For information using ON-Bar to back up storage spaces and logical logs, refer to the “Backup and Restore Guide.”*

Freeing a Log File with Status A

If a log file is newly added (status A), create a level-0 backup of the root dbspace and the dbspace that contains the log file to activate the log file and make it available for use.

Freeing a Log File with Status U

If a log file contains records but is not yet backed up (status U), back up the file using the backup tool that you usually use.

If backing up the log file does not change the status to free (F), its status changes to either U-B or U-B-L. See “Freeing a Log File with Status U-B” on page 21-14 or “Freeing a Log File with Status U-B-L” on page 21-15.

Freeing a Log File with Status U-B

If a log file is backed up but still in use (status U-B), some transactions in the log file are still under way or it contains the oldest update which is required for fast recovery.

To free a backed up log file that is in use

1. If you do not want to wait until the transactions complete, take the database server to quiescent mode. See "Immediately from On-Line to Quiescent" on page 9-8. Any active transactions are rolled back.
2. Because a log file with status U-B might contain the oldest update, you must use the **onmode -c** command to force a full checkpoint.

A log file that is backed up but *not* in use (status U-B) does not need to be freed. In the following example, log 34 does not need to be freed but logs 35 and 36 do. Log 35 contains the last checkpoint and log 36 is backed up but still in use.

```
34 U-B--   Log is used, backed up, and not in use
35 U-B-L   Log is used, backed up, contains last checkpoint
36 U-B--   Log is used, backed up, currently in use
37 U-C--   This is the current log file, not backed up
```

Freeing a Log File with Status U-C or U-C-L

If you want to free the current log file (status C), follow these steps:

1. Execute the following command:

   ```
   % onmode -l
   ```

   (Be sure to type a lowercase L on the command line, not a number 1.) This command switches the current log file to the next available log file.
2. Back up the original log file with the backup tool that you usually use.

   After all full log files are backed up, you are prompted to switch to the next available logical-log file and back up the new current log file. You do not need to do this because you just switched to this log file.

After you follow these steps, if the log file now has status U-B or U-B-L, refer to "Freeing a Log File with Status U-B" on page 21-14 or "Freeing a Log File with Status U-B-L."

Freeing a Log File with Status U-B-L

If a log file is backed up and all transactions within it are closed, but the file is not free (status U-B-L), this logical-log file contains the most-recent checkpoint record.

To free log files with a status U-B-L, the database server must create a new checkpoint. You can execute the following command to force a checkpoint:

```
onmode -c
```

Switching to the Next Logical-Log File

Switch to the next logical-log file before the current log file becomes full to back up the current log.

The database server can be in on-line mode to make this change. Execute the following command to switch to the next available log file:

```
onmode -l
```

The change takes effect immediately. (Be sure that you type a lowercase L on the command line, not a number 1.)

Monitoring Logging Activity

This section discusses how to monitor the logical-log files. For information on monitoring the logical-log buffers, see "Monitoring Physical and Logical Logging Activity" on page 23-5.

Monitor the logical-log files to determine the total available space (in all the files), the space available in the current file, and the status of a file (for example, whether the log has been backed up yet). This information is important for logical-log management.

Monitoring the Logical Log for Fullness

When the database server is blocking to preserve log space for administrative tasks, the **onstat** utility displays the following message just after its banner line:

```
Blocked: LBU
```

Suppose every log except the last one is full.

In these circumstances, the second line of any of the **onstat** options appear as shown in the following example:

```
Blocked: LBU
```

To unblock the database server, force a full checkpoint with the **onmode -c** command or a fuzzy checkpoint with the **onmode -c fuzzy** command.

Using Command-Line Utilities

You can use the following command-line utilities to monitor logical-log files.

onstat -l

The **onstat -l** utility display consists of the following three sections: physical-log information, logical-log information (general), and information on the individual logical-log files.

The third section contains the following information for each logical-log file:

- The address of the logical-log file descriptor
- The logical-log file logid number
- Status flags that indicate the status of each log

  Flags indicate whether the log is free, backed up, current, and so on.
- The unique ID of the log file
- The beginning page of the file
- The size of the file in pages, the number of pages used, and the percentage of pages used

For information on the **onstat -l** option, see the utilities chapter in the *Administrator's Reference*. Figure 21-1 shows sample output.

```
  ...
  address   number    flags     uniqid    begin         size      used     %used
  846640    1         F------   0         100233         250         0      0.00
  84665c    2         F------   0         10032d         250         0      0.00
  846678    3         U---C-L   3         100427         250       175     70.00
  846694    4         F------   0         100521         250         0      0.00
  8466b0    5         F------   0         10061b         250         0      0.00
```

Figure 21-1
onstat -l Output Showing Logical-Log File Status

onutil CHECK RESERVED

The database server stores logical-log file information in the reserved pages dedicated to checkpoint information. Because the database server updates this information only during a checkpoint, it is not as recent as the information that the **onstat -l** option displays. For more details on using these options to display reserved-page information, see the utilities chapter in the *Administrator's Reference*.

You can view the checkpoint reserve pages with the **onutil** CHECK RESERVED command. Figure 21-2 shows sample output.

```
   ...
   Log file number                1
   Log file flags                 0
   Time stamp                     6964
   Date/Time file filled          07/28/99 14:48:32
   Unique identifier              0
   Physical location              100233
   Log size                       250
   Number pages used              0
   ...
```

Figure 21-2
onutil CHECK RESERVED Output Containing Logical-Log File Information

Using SMI Tables

Query the **syslogs** table to obtain information on logical-log files. This table contains a row for each logical-log file. The columns are as follows.

Column	Description
number	Identification number of the logical-log file
uniqid	Unique ID of the log file
size	Size of the file in pages
used	Number of pages used
is_used	Flag that indicates whether the log file is being used
is_current	Flag that indicates whether the log file is current
is_backed_up	Flag that indicates whether the log file has been backed up
is_new	Flag that indicates whether the log file has been added since the last storage-space backup
is_archived	Flag that indicates whether the log file has been written to the archive tape
is_temp	Flag that indicates whether the log file is flagged as a temporary log file

Monitoring Log-Backup Status

To monitor the status of the logs and to see which logs have been backed up, use the **onstat -l** command. A status flag of B indicates that the log has been backed up.

To monitor the status of the logs on different coservers, use the **xctl onstat -l** command-line utility. This command allows you to see the status of every log file on every coserver.

To monitor the backup status of both logs and dbspaces, execute the **onstat -g bus** option. Figure 21-3 shows sample output.

```
Backup scheduler sessions
=========================

Session "Log backup 1" state SUSPENDED error 0
Session "Log backup 2" state SUSPENDED error 0
    LOG(log.2.11) BACKUP,RUNNING
Session "leia_tli24062" state WAITING error 0
    LOG(log.2.11) BACKUP,RUNNING
    LOG(log.1.16) BACKUP,READY
```

Figure 21-3
xctl onstat -g bus Output

The *Backup Scheduler* schedules backup and restore sessions. In Figure 21-3, ON-Bar is backing up logical log, **log.2.11**. The 2 in the log name refers to the coserver number and 11 refers to the logid. The other logical log, **log.1.16** on coserver 1, is ready to be backed up. Session **leia_tli24062** is the name of the backup session.

The suspended log backup sessions mean that LOG_BACKUP_MODE is set to MANUAL or that the user turned off continuous logical-log backup. When a logical log fills, it is ready to be backed up. The Backup Scheduler places it in the queue until the user starts a backup. For more information on LOG_BACKUP_MODE, see the *Backup and Restore Guide* and the chapter on configuration parameters in the *Administrator's Reference*.

Displaying Logical-Log Records

Use the **onlog** utility to display and interpret logical-log records. For information on using **onlog**, see the utilities chapter in the *Administrator's Reference*.

Chapter

Physical Logging

22

In This Chapter

This chapter defines the terms and explains the concepts that you need to know to perform effectively the tasks described in Chapter 23, "Managing the Physical Log." The chapter covers the following topics:

- Physical logging and the purposes it serves
- Some guidelines for the size and location of the physical log
- Details of the physical-logging process

Physical Logging

Physical logging is the process of storing the pages that the database server is going to change before the changed pages are actually recorded. Before the database server modifies certain pages in the shared-memory buffer pool, it stores an unmodified copy of the page (called a *before-image*) in the physical-log buffer in shared memory.

The *physical log* is a set of contiguous disk pages where the database server stores before-images.

The database server maintains the before-image page in the physical-log buffer in shared memory for those pages until one or more page cleaners flush the pages to disk. Once a checkpoint occurs, the database server empties the physical log (except in the special circumstances explained in "Limit to the Size of the Physical Log" on page 22-6). For more information on checkpoints, see "Checkpoints" on page 24-4.

Important: *The database server no longer logs the before-images for fuzzy operations in the physical log. It still tracks these updates in the logical log. For a definition of fuzzy operations, see "Fuzzy Checkpoint" on page 24-5.*

Purpose of Physical Logging

This seemingly odd activity of storing copies of pages before they are changed ensures that the unmodified pages are available in case the database server fails or the backup procedure needs them to provide an accurate snapshot of the database server data. These snapshots are potentially used in two activities: fast recovery and the database server backup.

Fast Recovery Use of Physically-Logged Pages

After a failure, the database server uses the before-images of pages modified by non-fuzzy operations in the physical log to restore these pages on the disk to their state at the last checkpoint. Then the database server uses the logical-log records to return all data to physical and logical consistency, up to the point of the most-recently completed transaction. Chapter 24, "Checkpoints and Fast Recovery," explains this procedure in more detail.

Backup Use of Physically-Logged Pages

When you perform a storage-space backup, the database server performs a full checkpoint and checks disk pages to see which should be backed up. If a backup is active, the database server sends physically-logged pages to the backup program. For more details, see the *Backup and Restore Guide* if you use ON-Bar.

Database Server Activity That Is Physically Logged

All dbspace page modifications except the following ones are physically logged:

- Pages that do not have a valid database server address

  This situation usually occurs when the page was used by some other database server or a table that was dropped.
- Pages that the database server has not allocated and that are located in a dbspace where no table has been dropped since the last checkpoint.
- Pages for fuzzy operations such as inserts, deletes, and updates.

In case of multiple modifications before the next checkpoint, only one before-image is logged in the physical log (the first before-image).

Storing all before-images of page modifications in the physical log might seem excessive. But the database server stores the before-images in the physical log only until the next checkpoint. To control the amount of data that the database server logs, you can tune the checkpoint interval configuration parameter CKPTINTVL.

Physical Logging and Simple Large Objects

The database server pages in the physical log can be any database server page, including simple large objects in tblspaces. Even overhead pages (such as chunk free-list pages) are copied to the physical log before data on the page is modified and flushed to disk.

Size and Location of the Physical Log

When you consider how large to make your physical log, you can begin by using the following formula to calculate an approximate size:

```
PHYSFILE = (connections * max_log_pages_per_crit_sect * 4 * pagesize) / 1024
```

This PHYSFILE value represents a maximum. For more information on monitoring and tuning the physical log, refer to the chapter on configuration effects on I/O utilization in your *Performance Guide*.

Variable in Formula	Description
connections	Maximum number of users that you expect to access the database server concurrently. If you set the NETTYPE parameter, connections is the sum of the values specified in the **users** field of each NETTYPE parameter in your ONCONFIG file.
max_log_pages_per_crit_sect	Maximum number of pages that the database server can physically log in a critical section.

(1 of 2)

Variable in Formula	Description
4	Necessary factor because the following part of the formula represents only 25 percent of the physical log: *connections * max_log_pages_per_crit_sect*
pagesize	System page size in bytes that you can obtain with **onutil** CHECK RESERVED.
1024	Necessary divisor because you specify PHYSFILE parameter in units of kilobytes

(2 of 2)

Limit to the Size of the Physical Log

Because a checkpoint logically empties the physical log when it becomes 75 percent full, it is unlikely that the log would become 100 percent full before the checkpoint completes. To assure further that the physical log does not become full during a checkpoint, take the following actions:

- Configure the database server according to the sizing guidelines for the physical log and the logical-log files.
- Fine-tune the size of the physical log by monitoring it during production activity.

Fuzzy checkpoints keep the physical log from filling up too quickly when applications are doing intensive updates. (See "Fuzzy Checkpoint" on page 24-5.) However, the physical log could still become full, as the following sections describe.

Physical-Log Overflow When Many Users Are in Critical Sections

Under normal processing, once a checkpoint is requested, and the checkpoint begins, all threads are prevented from entering critical sections of code. (See "Critical Sections" on page 24-4.) However, threads *currently* in critical sections can continue processing. The physical log can become full if many threads in critical sections are processing work *and* if the space that remains in the physical log is very small. The many writes that are performed as threads complete their critical section processing could conceivably cause the physical log to become full.

Effect of Checkpoints on the Physical-Log Size

Fuzzy checkpoints keep the physical log from filling up too quickly when applications are doing intensive updates. You can reduce the size of the physical log when applications require less intensive updates or when updates tend to cluster within the same pages. You can decrease the size of the physical log if you intend to use physical-log fullness to trigger checkpoints.

If you increase the checkpoint interval or anticipate increased activity, consider increasing the size of the physical log. For more information, see the chapter on effects of configuration on I/O activity in your *Performance Guide*.

Physical-Log Overflow When Transaction Logging Is Turned Off

The physical log can overflow if you use simple large objects in a nonlogging table. Consider the following example about simple large objects in dbspaces stored in a logging table.

When the database server processes these simple large objects, each portion of the simple large object that the database server stores on disk can be logged separately, allowing the thread to exit the critical sections of code between each portion. However, if logging is turned off, the database server must carry out all operations on the simple large object in one critical section. If the simple large object is large, and the physical log small, this scenario can cause the physical log to become full. If this situation occurs, the database server sends the following message to the message log:

```
Physical log file overflow
```

The database server then initiates a shutdown. For the suggested corrective action, refer to this message in your message log.

Physical-Log Overflow During Rollback of a Long Transaction

This same unlikely scenario could occur during the rollback of a long transaction after the second long-transaction high-water mark, LTXEHWM, is reached. (See "Logical Log and Long Transactions" on page 20-15.) After the LTXEHWM is reached, and after all threads have exited critical sections, only the thread that is performing the rollback has access to the physical and logical logs. However, the writes that are performed as threads complete their processing could conceivably fill the physical log during the rollback if the following conditions occur simultaneously:

- Many threads were in critical sections.
- The space remaining in the physical log was very small at the time that the LTXEHWM was reached.

Location of the Physical Log

When the database server initializes disk space, it places the logical-log files and the physical log in the root dbspace. You have no initial control over this placement. To improve performance (specifically, to reduce the number of writes to the root dbspace and minimize disk contention), you can move the physical log out of the root dbspace to another dbspace, preferably on a disk that does not contain active tables or the logical-log files.

The physical log is located in the dbspace specified by the ONCONFIG parameter PHYSDBS. (For information on PHYSDBS, see the chapter on configuration parameters in the *Administrator's Reference*.) Change PHYSDBS only if you decide to move the physical-log file from the root dbspace. (See "Changing the Physical-Log Location and Size" on page 23-3.)

Because the physical log is critical, Informix recommends that you mirror the dbspace that contains the physical log.

Details of Physical Logging

This section describes the details of physical logging. It is provided to satisfy your curiosity; you do not need to understand the information here in order to manage your physical log.

The database server performs physical logging in the following six steps:

1. Reads the data page from disk to the shared-memory page buffer (if the data page is not there already)
2. Copies the unchanged page to the physical-log buffer
3. Reflects the change in the page buffer after an application modifies data
4. Flushes the physical-log buffer to the physical log on disk
5. Flushes the page buffer and writes it back to disk
6. When a checkpoint occurs, flushes the physical-log buffer to the physical log on disk and empties the physical log

The paragraphs that follow explain each step in detail.

Page Is Read into the Shared-Memory Buffer Pool

When a session requests a row, the database server identifies the page on which the row resides and attempts to locate the page in the database server shared-memory buffer pool. If the page is not already in shared memory, it is read into the resident portion of the database server shared memory from disk.

A Copy of the Page Buffer Is Stored in the Physical-Log Buffer

If the before-image of a modified page is stored in the physical-log buffer, it is eventually flushed from the physical-log buffer to the physical log on disk. If the same page is modified again before the next checkpoint, it does not require another before-image to be stored in the physical-log buffer. (Fuzzy operations do not physically log before-images of pages.) The before-image of the page plays a critical role in restoring data and fast recovery. For more details, see "Physical-Log Buffer" on page 13-24.

Change Is Reflected in the Data Buffer

The database server reflects changes to the data in the shared-memory data buffer. Data from the application is passed to the database server. After a copy of the unchanged data page is stored in the physical-log buffer, the new data is written to the page buffer already acquired.

Physical-Log Buffer Is Flushed to the Physical Log

The database server flushes the physical-log buffer before it flushes the data buffer to ensure that a copy of the unchanged page is available until the changed page is copied to disk. The before-image of the page is no longer needed after a checkpoint occurs. For more details, see "Flushing the Physical-Log Buffer" on page 13-46.

Page Buffer Is Flushed

After the physical-log buffer is flushed, the shared-memory page buffer is flushed to disk (such as during a checkpoint), and the data page is written to disk. Only non-fuzzy pages are flushed to disk during a fuzzy checkpoint. For conditions that lead to the flushing of the page buffer, see "Flushing Data to Disk" on page 13-45.

When a Checkpoint Occurs

A checkpoint can occur at any point in the physical-logging process. The database server performs two types of checkpoints: *full* and *fuzzy*. After a full checkpoint occurs, all modified pages in shared memory are flushed to disk. The database server is physically consistent because all changes to the data since the prior checkpoint are recorded on disk and in the logical log. For information, see "Full Checkpoint" on page 24-5

After a fuzzy checkpoint occurs, the database server might not be physically consistent, but the checkpoint completes much quicker and does not tie up the database server during heavy update activity. All changes to the data since the last full checkpoint or fast recovery are recorded in the logical log. For information, see "Fuzzy Checkpoint" on page 24-5.

How the Physical Log Is Emptied

The database server manages the physical log as a circular file, constantly overwriting unneeded data. The checkpoint procedure empties the physical log by resetting a pointer in the physical log that marks the beginning of the next group of required before-images.

Chapter

Managing the Physical Log

23

In This Chapter

This chapter describes procedures for changing the location and size of the physical log and for monitoring the physical log. For background information about the physical log, see Chapter 22, "Physical Logging."

Changing the Physical-Log Location and Size

To change your physical-log location or size, use a text editor to edit the ONCONFIG file.

Log in as user **informix** or **root** when you make the changes. The following sections describe each of these methods.

For any of the three methods, to activate the changes to the size or location of the physical log as soon as you make them, shut down and restart the database server to reinitialize shared memory. If you use **onparams**, you can reinitialize shared memory in the same step.

Create a complete level-0 backup immediately after you reinitialize shared memory. This storage-space backup is critical for database server recovery.

Reasons to Change the Physical-Log Location and Size

You can move the physical-log file to try to improve performance. When the database server initializes disk space, it places the disk pages allocated for the logical log and the physical log in the root dbspace. You might improve performance by moving the physical log, the logical-log files, or both to other dbspaces.

For advice on where to place the physical log, see "Location of the Physical Log" on page 22-8. For advice on sizing the physical log, see"Size and Location of the Physical Log" on page 22-5. To obtain information about the physical log, see "Monitoring Physical and Logical Logging Activity" on page 23-5.

Preparing to Make the Changes

The space allocated for the physical log must be contiguous. If you move the physical log to a dbspace without adequate contiguous space, or if you increase the log size beyond the available contiguous space, a fatal shared-memory error occurs when you attempt to reinitialize shared memory with the new values. If this error occurs, resize the physical log, or choose another dbspace with adequate contiguous space and then reinitialize shared memory.

Checking For Adequate Contiguous Space

You can check if adequate contiguous space is available with the CHECK SPACE option of the **onutil** utility. For more information on the **onutil** CHECK SPACE command, see the utilities chapter in the *Administrator's Reference*.

For more information, see "Monitoring Chunks" on page 16-34.

Using a Text Editor to Change Physical-Log Location or Size

You can change the physical-log location and size by editing the ONCONFIG file while the database server is in on-line mode.

Parameter	Description
PHYSFILE	Specifies the size of the physical log file in kilobytes
PHYSDBS	Moves the physical log to the specified dbspace
PHYSSLICE	Moves the physical log to the specified dbspace or dbslice on each coserver

The changes do not take effect until you shut down and restart the database server. Then, create a level-0 backup immediately to ensure that all recovery mechanisms are available.

For information on PHYSSLICE, PHYSFILE, and PHYSDBS, see the chapter on configuration parameters in the *Administrator's Reference*.

Monitoring Physical and Logical Logging Activity

This section discusses monitoring the physical-log file, physical-log buffers, and logical-log buffers.

Monitor the physical log to determine the percentage of the physical-log file that gets used before a checkpoint occurs. This information allows you to find the optimal size of the physical-log file. It should be large enough that the database server does not have to force checkpoints too frequently and small enough to conserve disk space and guarantee fast recovery.

Monitor physical-log and logical-log buffers to determine if they are the optimal size for the current level of processing. The important statistic to monitor is the pages-per-disk-write statistic. For more information on tuning the physical-log and logical-log buffers, see your *Performance Guide*.

Using Command-Line Utilities

You can use the following command-line utilities to obtain information about the physical-log file.

onstat -l

The first line of the **onstat -l** output displays the following information for each physical-log buffer:

- The number of buffer pages used (**bufused**)
- The size of each physical log buffer in pages (**bufsize**)
- The number of pages written to the buffer (**numpages**)
- The number of writes from the buffer to disk (**numwrits**)
- The ratio of pages written to the buffer to the number of writes to disk (**pages/IO**)

The second line of the **onstat -l** output displays the following information about the physical log:

- The page number of the first page in the physical-log file (**phybegin**)
- The size of the physical-log file (**physize**)
- The current position in the log where the next write occurs (**physpos**)
- The number of pages in the log that have been used (**phyused**)
- The percentage of the total physical-log pages that have been used (**%used**)

The third line of the **onstat -l** output displays the following information about each logical-log buffer:

- The number of buffer pages used (**bufused**)
- The size of each logical-log buffer in pages (**bufsize**)
- The number of records written to the buffer (**numrecs**)
- The number of pages written to the buffer (**numpages**)
- The number of writes from the buffer to disk (**numwrits**)
- The ratio of records to pages in the buffer (**recs/pages**)
- The ratio of pages written to the buffer to the number of writes to disk (**pages/IO**)

Figure 23-1 shows sample output from the **onstat -l** option that contains the relevant fields.

```
Physical Logging
Buffer bufused  bufsize  numpages numwrits pages/io
  P-2  0        16       110      10       11.00
      phybegin physize  phypos   phyused  %used
      10003f   500      233      0        0.00

Logical Logging
Buffer bufused  bufsize  numrecs  numpages numwrits recs/pages pages/io
  L-1  0        16       3075     162      75       19.0       2.2
...
```

Figure 23-1
onstat -l Output That Shows Information About the Physical and Logical Logs

onutil CHECK RESERVED

The database server stores the physical-log file information in those reserved pages dedicated to checkpoint information (PAGE_1CKPT and PAGE_2CKPT). You can view the checkpoint reserve pages with the **onutil** CHECK RESERVED command. The reserve pages contain the state of the physical log at the last checkpoint. Figure 23-2 shows an example of the relevant output.

```
Validating Informix database server reserved pages - PAGE_1CKPT & PAGE_2CKPT
          Using check point page PAGE_2CKPT.

    Time stamp of checkpoint       16024
    Time of checkpoint             07/30/99 09:34:33
    Physical log begin address     10003f
    Physical log size              500
    Physical log position at Ckpt  e9
...
```

Figure 23-2
onutil CHECK RESERVED Output That Includes Physical-Log Information

Using SMI Tables

Query the **sysprofile** table to obtain statistics on the physical-log and logical-log buffers. The following rows contain the relevant statistics.

Row	Description
plgpagewrites	Number of pages written to the physical-log buffer
plgwrites	Number of writes from the physical-log buffer to the physical log file
llgrecs	Number of records written to the logical-log buffer
llgpagewrites	Number of pages written to the logical-log buffer
llgwrites	Number of writes from the logical-log buffer to the logical-log files

Chapter

Checkpoints and Fast Recovery

24

In This Chapter

This chapter describes how the database server achieves data consistency through checkpoints and the fast-recovery feature. Read this chapter if you are interested in learning how checkpoints and fast recovery work.

This chapter covers the following topics:

- How the database server achieves data consistency
- Critical sections
- Checkpoints to achieve data consistency
- Time stamps to synchronize events
- Fast recovery
- Monitoring checkpoints

How the Database Server Achieves Data Consistency

The database server uses the following three procedures to ensure that data destined for disk is actually recorded intact on disk:

- Critical sections
- Checkpoints
- Time stamps

These procedures ensure that multiple, logically related writes are recorded as a unit; that data in shared memory is periodically made consistent with data on disk; and that a buffer page that is written to disk is actually written in entirety.

Critical Sections

A *critical section* of code makes a set of disk modifications that must be performed as a single unit; either all the modifications must occur, or none can occur.

A thread that is in a critical section is holding shared-memory resources. Within the space of the critical section, the database server cannot determine which shared-memory resources should be released and which changes should be undone to return all data to a consistent point. Therefore, if a virtual processor is terminated while a thread is in a critical section, the database server takes the following two steps to ensure that all data is returned to the last known point of consistency:

- The database server aborts immediately.
- The database server initiates fast recovery the next time that it is initialized.

Fast recovery is the procedure that the database server uses to restore the physical and logical consistency of data quickly, up to and including the last record in the logical log. For a description of fast recovery, refer to "Fast Recovery" on page 24-13.

Checkpoints

The database server performs two types of checkpoints: full checkpoints (also known as *sync* checkpoints) and fuzzy checkpoints. The term *checkpoint* refers to the point in the database server operation when the pages on disk are synchronized with the pages in the shared-memory buffer pool.

The database server generates at least one checkpoint for each span of the logical-log space to guarantee that it has a checkpoint at which to begin fast recovery.

Although the database server performs checkpoints automatically, you can initiate one manually or control how often the database server checks to see if a checkpoint is needed. You can specify the checkpoint interval in the CKPTINTVL configuration parameter. To reduce the amount of work required at checkpoint, lower the LRU_MAX and LRU_MIN values. For more information about CKPTINTVL, LRU_MAX, and LRU_MIN, see the chapter on configuration parameters in the *Administrator's Reference*. For information on monitoring and tuning checkpoint parameters, see your *Performance Guide*.

Full Checkpoint

In a *full checkpoint*, the database server flushes all modified pages in the shared-memory buffer pool to disk. When a full checkpoint completes, all physical operations are complete, the MLRU queue is empty, and the database server is said to be physically consistent.

Fuzzy Checkpoint

In a *fuzzy checkpoint*, the database server does not flush the modified pages in the shared-memory buffer pool to disk for certain types of operations, called *fuzzy operations*. When a fuzzy checkpoint completes, the checkpointed pages might not be consistent with each other because the database server does not flush all data pages to disk. When necessary, the database server performs a full checkpoint to ensure the physical consistency of all data on disk.

Fuzzy Operations

The following commonly used operations are fuzzy:

- Inserts
- Updates
- Deletes

These following operations are *nonfuzzy*:

- Rows that contain simple large objects (TEXT and BYTE data types)
- Table alters
- Operations that modify index keys

The database server flushes *all* the modified data pages for nonfuzzy operations to disk during a fuzzy checkpoint in the same way as for a full checkpoint.

Write-Ahead Logging and Fast Recovery

Fuzzy checkpoint depends on write-ahead logging for fast recovery to work correctly. *Write-ahead logging* means that the logical-log records representing changes to fuzzy data must be on disk before the changed data replaces the previous version of the data on disk. Fast recovery begins with the oldest update not yet flushed to disk rather than with the previous checkpoint.

Fuzzy Checkpoints Improve Performance

Fuzzy checkpoints are much faster than full checkpoints and improve transaction throughput. Because the database server does not log fuzzy operations in the physical log, the physical log does not fill as quickly, and checkpoints occur less often. For example, if you are inserting and updating a lot of data, checkpoints occur less frequently and are shorter.

The database server skips a full checkpoint if all data is physically consistent when the checkpoint interval expires. It skips a fuzzy checkpoint only if no pages have been dirtied since the last checkpoint.

For information on improving checkpoint performance, see the chapter about configuration impacts on I/O in your *Performance Guide*. For information about the physical log, see Chapter 22, “Physical Logging”

Events That Initiate a Fuzzy Checkpoint

Usually, when the database server performs an automatic checkpoint, it is a fuzzy checkpoint. Any one of following conditions initiates a fuzzy checkpoint:

- The checkpoint interval, specified by the configuration parameter CKPTINTVL, has elapsed, and one or more modifications have occurred since the last checkpoint.
- The physical log on disk becomes 75 percent full.
- The database server detects that the next logical-log file to become current contains the most recent checkpoint record.
- Certain administrative tasks, such as adding a chunk or a dbspace, take place.

Events That Initiate a Full Checkpoint

In the following situations, the database server performs a full checkpoint to ensure the physical consistency of all data on disk:

- When you issue **onmode -ky** to shut down the database server
- When you initiate a checkpoint from the command line with **onmode -c**
- When you convert the database server to a newer version or revert to a previous version
- When you perform a backup or restore using ON-Bar

  The backup tool performs a full checkpoint automatically to ensure the physical consistency of all data before it writes it to the backup media.
- At the end of fast recovery or full recovery

- If the database server is about to switch to the next free log and the log following the free log contains the oldest update

  For example, suppose four logical-log files have the status shown in the following list. The database server forces a full checkpoint when it switches to logical-log file 3 if the logical-log file 4 has the oldest update. The full checkpoint advances the oldest update to logical-log file 3.

logid	Logical-Log File Status
1	U-B----
2	U---C--
3	F
4	U-B---L

  The database server performs a full checkpoint to prevent problems with fast recovery of old log records.

For a list of situations in which you should initiate a full checkpoint, see the following section.

Forcing a Full Checkpoint

You might want to force a full checkpoint for any of the following reasons, as well as others:

- You should initiate a full checkpoint to free a logical-log file that contains the most recent checkpoint record and that is backed up but not yet released (**onstat -l** status of `U-B-L` or `U-B`).
- You should initiate a full checkpoint before you issue **onmode -sy** to place the database server in quiescent mode.
- You have just finished building a large index. If the database server terminates before the next checkpoint, the index build will restart the next time that you initialize the database server.
- You are about to attempt a system operation that might interrupt the database server. If a checkpoint has not occurred for a long time, fast recovery could take longer than you want.

- Foreground writes are taking more resources than you want. You can manually force a checkpoint to bring this down to zero for a while.
- You should initiate a full checkpoint before you execute **dbexport** or unload a table. The full checkpoint ensures the physical consistency of all data before you export or unload it.

To force a checkpoint, execute the following command from the command line:

```
onmode -c
```

Forcing a Fuzzy Checkpoint

To force a fuzzy checkpoint, execute the following command:

```
onmode -c fuzzy
```

Sequence of Events in a Checkpoint

The following section outlines the main events that occur during a checkpoint once a user thread raises the checkpoint-requested flag. This section also notes the differences between full and fuzzy checkpoints:

1. The database server prevents user threads from entering critical sections.
2. The logical-log buffer is flushed to the current logical-log file on disk.
3. The page-cleaner thread flushes the physical-log buffer.
4. In a fuzzy checkpoint, the page-cleaner threads flush modified pages for nonfuzzy operations in the buffer pool to disk.

   In a full checkpoint, the page-cleaner threads flush all modified pages in the buffer pool to disk.
5. The checkpoint thread writes a checkpoint record to the logical-log buffer.
6. The physical log on disk is logically emptied. (Current entries can be overwritten).

User Threads Cannot Enter a Critical Section

This step is the same for both fuzzy and full checkpoints. Once the checkpoint-requested flag is set, user threads are prevented from entering portions of code that are considered critical sections. User threads that are within critical sections of code are permitted to continue processing to the end of the critical sections.

Logical-Log Buffer Is Flushed to the Logical-Log File on Disk

This step is the same for both fuzzy and full checkpoints. Next, the logical-log buffer is flushed to the logical-log file on disk.

Page-Cleaner Thread Flushes the Physical-Log Buffer

After all threads have exited from critical sections, the page-cleaner thread resets the shared-memory pointer from the current physical-log buffer to the other buffer and flushes the buffer. After the buffer is flushed, the page-cleaner thread updates the time stamp that indicates the most recent point at which the physical-log buffer was flushed.

Page-Cleaner Threads Flush Modified Pages in the Buffer Pool

In a fuzzy checkpoint, the page-cleaner threads flush modified pages for nonfuzzy operations in the buffer pool to disk. They do not flush modified pages for fuzzy operations (inserts, deletes, updates) to disk. Figure 24-1 shows how the database server writes only the nonfuzzy pages to disk. The shaded squares, marked **F**, represent the fuzzy pages.

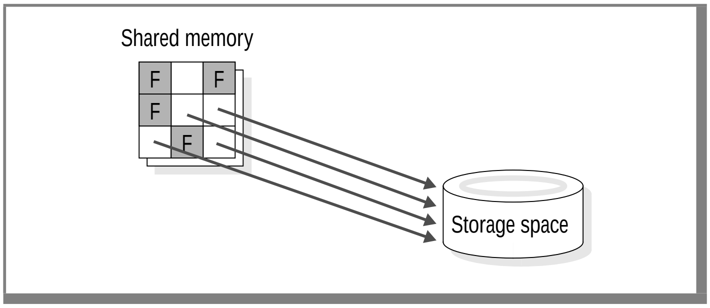


Figure 24-1
Selectively Writing Modified Pages from Shared Memory to Disk

In a full checkpoint, the page cleaners flush all modified pages in the shared-memory buffer pool to disk. This flushing is performed as a chunk write.

Checkpoint Thread Writes Checkpoint Record

This step is the same for both fuzzy and full checkpoints. The page-cleaner thread writes a *checkpoint-complete* record to the logical-log buffer after the modified pages have been written to disk.

In a fuzzy checkpoint, the checkpoint thread also writes a dirty-pages table (DPT) record to the logical-log buffer. For more information, see the chapter on logical-log record types in the *Administrator's Reference*.

Physical Log Is Logically Emptied

This step is the same for both fuzzy and full checkpoints. After the checkpoint-complete record is written to disk, the physical log is logically emptied, meaning that current entries in the physical log can be overwritten.

Backup and Restore Considerations

If you perform a backup, the database server performs a full checkpoint and flushes all changed pages, including those for fuzzy operations, to the disk. If you perform a restore, the database server reapplies all logical-log records.

Important: *Because the logical log contains records of fuzzy operations not yet written to disk, you must back up the logical logs regularly.*

For information on ON-Bar, see the *Backup and Restore Guide*.

Time Stamps

The database server uses a time stamp to identify a time when an event occurred relative to other events of the same kind. The time stamp is not a literal time that refers to a specific hour, minute, or second. It is a 4-byte integer that the database server assigns sequentially. When the database server compares two time stamps, its algorithm accounts for the possibility that wraparound has occurred.

Time Stamps on Disk Pages

Each disk page has one time stamp in the page header and a second time stamp in the last 4 bytes on the page. The page-header and page-ending time stamps are synchronized after each write, so they should be identical when the page is read from disk. Each read compares the time stamps as a test for data consistency. If the test fails, an error is returned to the user thread, indicating either that the disk page was not fully written to disk or that the page has been partially overwritten on disk or in shared memory. For a description of the content of a dbspace page, refer to dbspace structure and storage in the chapter on disk structures and storage in the *Administrator's Reference*.

Time Stamps on Logical-Log Pages

Each logical-log record contains a time stamp, a 4-byte integer. During a restore, the database server uses these time stamps to determine where to start the logical restore.

For fuzzy operations, the database server uses the same time stamp for both the log record and disk page. During fast recovery, the database server compares the disk-page and log time stamps to determine which log record to apply. After fast recovery applies the log record to the dirty page, it copies the time stamp from the log record to the time stamp for the disk page.

Fast Recovery

Fast recovery is an automatic, fault-tolerant feature that the database server executes every time that it moves from off-line to quiescent mode or from off-line to on-line mode. You do not need to take any administrative actions for fast recovery; it is an automatic feature.

The fast-recovery process checks if, the last time that the database server went off-line, it did so in uncontrolled conditions. If so, fast recovery returns the database server to a state of physical and logical consistency, as described in "Details of Fast Recovery After A Full Checkpoint" on page 24-14.

If the fast-recovery process finds that the database server came off-line in a controlled manner, the fast-recovery process terminates, and the database server moves to on-line mode.

Need for Fast Recovery

Fast recovery restores the database server to physical and logical consistency after any failure that results in the loss of the contents of memory for the database server. Such failures are usually caused by system failures. System failures do not damage the database but instead affect transactions that are in progress at the time of the failure.

Fast recovery addresses the following kinds of system failure:

- The database server is processing tasks for more than 40 users.
- Dozens of transactions are in process.
- Without warning, the operating system fails.

How does the database server bring itself to a consistent state again? What happens to ongoing transactions? The answer to both questions is fast recovery.

Situations When Fast Recovery Is Initiated

Every time that the administrator brings the database server to quiescent mode or on-line mode from off-line mode, the database server checks to see if fast recovery is needed.

As part of shared-memory initialization, the database server checks the contents of the physical log. The physical log is empty when the database server shuts down under control. The move from on-line mode to quiescent mode includes a checkpoint, which flushes the physical log. Therefore, if the database server finds pages in the physical log, the database server clearly went off-line under uncontrolled conditions, and fast recovery begins.

Fast Recovery and Buffered Logging

If a database uses buffered logging (as described in "Buffered Transaction Logging" on page 18-10), some logical-log records associated with committed transactions might not be written to the logical log at the time of the failure. If this occurs, fast recovery cannot restore those transactions. Fast recovery can restore only transactions with an associated COMMIT record stored in the logical log on disk. (For this reason, buffered logging represents a trade-off between performance and data vulnerability.)

Fast Recovery and No Logging

For databases or tables that do not use logging, fast recovery restores the database to its state at the time of the most recent checkpoint. All changes made to the database since the last checkpoint are lost. All fuzzy operations (inserts, deletes, updates) not yet flushed to disk are also lost.

Details of Fast Recovery After A Full Checkpoint

Fast recovery works differently depending on whether the previous checkpoint was a full or fuzzy checkpoint. This section discusses fast recovery after a full checkpoint.

Fast recovery returns the database server to a consistent state as part of shared-memory initialization. The consistent state means that all committed transactions are restored, and all uncommitted transactions are rolled back.

Fast recovery is accomplished in the following two stages:

- The database server uses the physical log to return to the most recent point of known *physical consistency,* the most recent checkpoint.
- The database server uses the logical-log files to return to *logical consistency* by rolling forward all committed transactions that occurred after the last checkpoint and rolling back all transactions that were left incomplete.

Fast recovery occurs in the following steps:

1. Use the data in the physical log to return all disk pages to their condition at the time of the most recent checkpoint.
2. Locate the most recent checkpoint record in the logical-log files.
3. Roll forward all logical-log records written after the most recent checkpoint record.
4. Roll back transactions that do not have an associated COMMIT or BEGIN COMMIT record in the logical log.

The paragraphs that follow describe each step in detail.

Returning to the Last-Checkpoint State

To accomplish the first step, returning all disk pages to their condition at the time of the most recent checkpoint, the database server writes the before-images stored in the physical log to shared memory and then back to disk. Each before-image in the physical log contains the address of a page that was updated after the checkpoint. When the database server writes each before-image page in the physical log to shared memory and then back to disk, changes to the database server data since the time of the most recent check-point are undone. Figure 24-2 on page 24-16 illustrates this step.

The database server is now physically consistent.

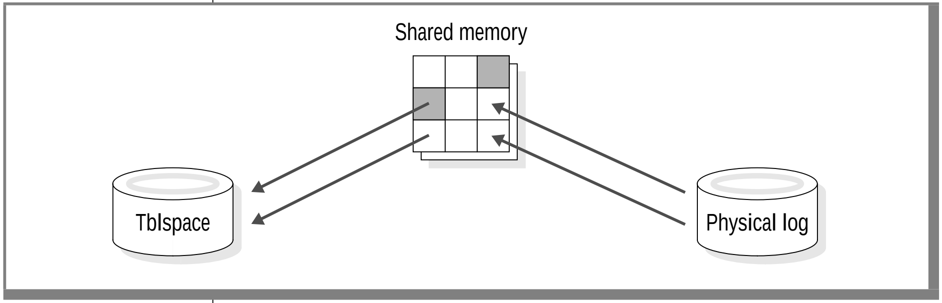


Figure 24-2
Writing All Remaining Before-Images in the Physical Log Back to Disk

Finding the Checkpoint Record in the Logical Log

In the second step, the database server locates the address of the most recent checkpoint record in the logical log. The most recent checkpoint record is guaranteed to be in the logical log on disk. Figure 24-3 illustrates this step.

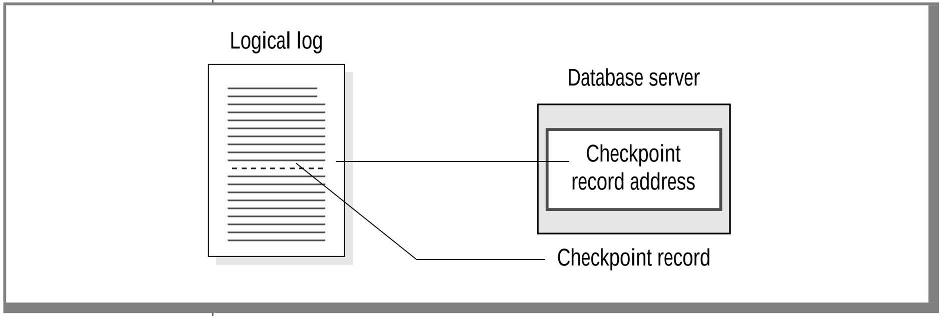


Figure 24-3
Locating the Most Recent Checkpoint Record in the Logical Log

Rolling Forward Logical-Log Records

The third step in fast recovery rolls forward the logical-log records that were written after the most recent checkpoint record. This action reproduces all changes to the databases since the time of the last checkpoint, up to the point at which the uncontrolled shutdown occurred. Figure 24-4 illustrates this step.

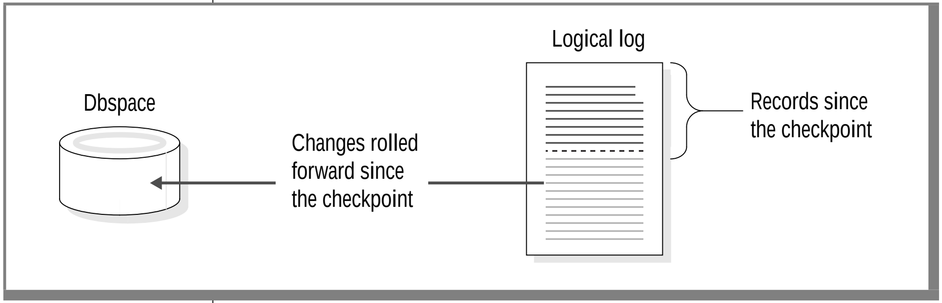

Logical log
Dbspace
Records since the checkpoint
Changes rolled forward since the checkpoint

Figure 24-4
Rolling Forward the Logical-Log Records Written Since the Most Recent Checkpoint

Rolling Back Incomplete Transactions

The final step in fast recovery rolls back all logical-log records for transactions that were not committed at the time the system failed. All databases are logically consistent because all committed transactions are rolled forward and all uncommitted transactions are rolled back.

Because one or more transactions possibly spanned several checkpoints without being committed, this rollback procedure might read backward through the logical log past the most recent checkpoint record. All logical-log files that contain records for open transactions are available to the database server because a log file is not freed until all transactions that it contains are closed. Figure 24-5 illustrates the rollback procedure. When fast recovery is complete, the database server goes to quiescent or on-line mode.

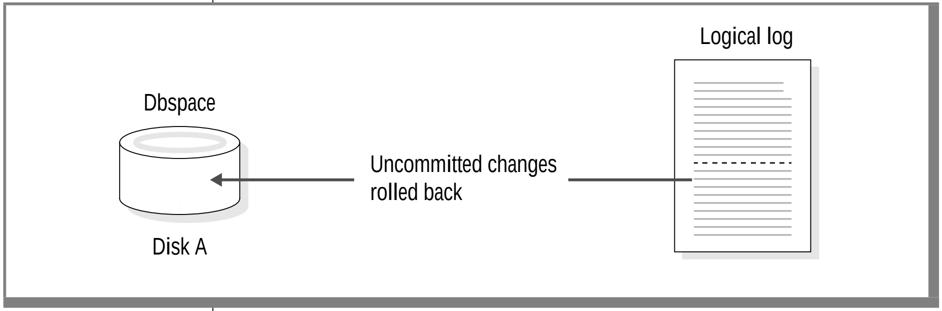


Figure 24-5
Rolling Back All Incomplete Transactions

Details of Fast Recovery After A Fuzzy Checkpoint

This section discusses fast recovery after a fuzzy checkpoint. Fast recovery is accomplished in the following stages:

- The database server uses the physical log to return to the most recent checkpoint. The database server might not be physically consistent at this point in fast recovery because fuzzy operations do not physically log the before-image of pages.
- The database server processes the logical-log records starting with the oldest update that has not yet been flushed to disk rather than starting with the previous checkpoint.
- The database server uses the logical-log files to return to *logical consistency* by rolling forward all committed transactions that occurred after the last checkpoint and rolling back all transactions that were left incomplete.

These stages can also be expressed as the following steps, which are described in detail in the paragraphs that follow:

1. Use the data in the physical log to return disk pages for nonfuzzy operations to their condition at the time of the most recent checkpoint.
2. Locate the oldest update in the logical-log that is not yet flushed to disk.
3. Apply the log records for fuzzy operations that occurred before the most recent checkpoint.
4. Roll forward all logical-log records written after the most recent checkpoint record.
5. Roll back transactions that do not have an associated COMMIT or BEGIN COMMIT record in the logical log.

Although fast recovery after a fuzzy checkpoint takes longer than after a full checkpoint, you can optimize it. For details, see your *Performance Guide*.

Returning to the Last-Checkpoint State for Nonfuzzy Operations

To accomplish the first step, returning all disk pages for nonfuzzy operations to their condition at the time of the most recent checkpoint, the database server writes the before-images stored in the physical log to shared memory and then back to disk. Each before-image in the physical log contains the address of a page that was updated after the checkpoint. When the database server writes each before-image page in the physical log to shared memory and then back to disk, changes to the database server data since the time of the most recent checkpoint are undone. Figure 24-6 illustrates this step.

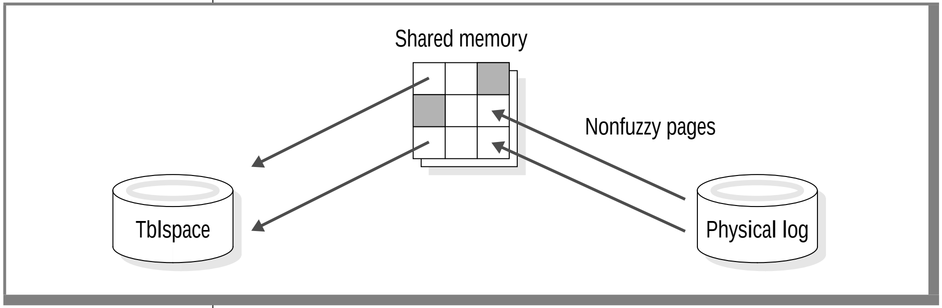


Figure 24-6
Writing Nonfuzzy Before-Images in the Physical Log Back to Disk

Pages on which fuzzy operations occurred are not physically consistent because the database server does not physically log their before-images. If the most recent checkpoint was a fuzzy checkpoint, the changed pages for fuzzy operations were not flushed to disk. The dbspace disk still contains the before-image of each page. To undo changes to these pages prior to the fuzzy checkpoint, the database server uses the logical log, as the next step describes.

Locating the Oldest Update in the Logical Log

In this step of fast recovery, the database server locates the oldest update record in the logical log that was not flushed to disk during the most recent checkpoint. The database server uses the log sequence numbers (LSN) in the logical log to find the oldest update record. The database server no longer starts fast recovery at the most recent checkpoint record.

Figure 24-7 shows that the oldest update in the logical log occurred several checkpoints ago and that all the log records are applied.

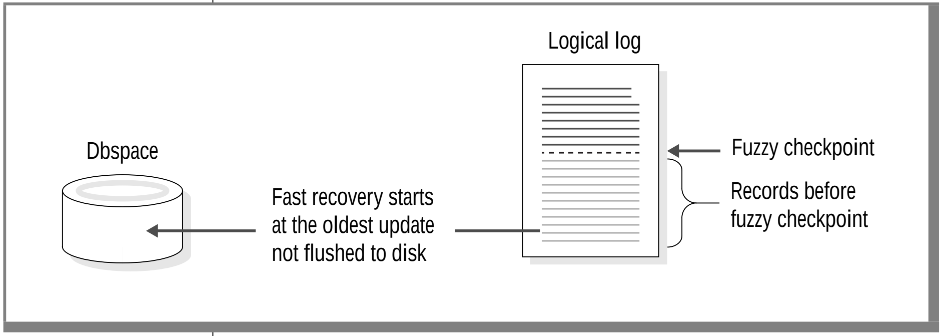

Logical log
Dbspace
Fuzzy checkpoint
Records before fuzzy checkpoint
Fast recovery starts at the oldest update not flushed to disk

Figure 24-7
Locating the Oldest Update Record in the Logical Log

You cannot free the logical log that contains the oldest update record until after the changes are recorded on disk. The database server automatically performs a full checkpoint to prevent problems with fast recovery of very old log records.

Applying the Log Records for Fuzzy Operations

In the second step, the database server processes the log records for fuzzy operations that occurred following the oldest update and before the last checkpoint. The log records that represent changes to data must be on disk before the changed data replaces the previous version on disk.

Log records for fuzzy operations are selectively redone, depending on whether the update has already been applied to the page. If the time stamp in the logical-log record is older than the time stamp in the disk page, the database server applies the record. Otherwise, the database server skips that record.

Figure 24-8 illustrates how the database server processes fuzzy records only prior to checkpoint.

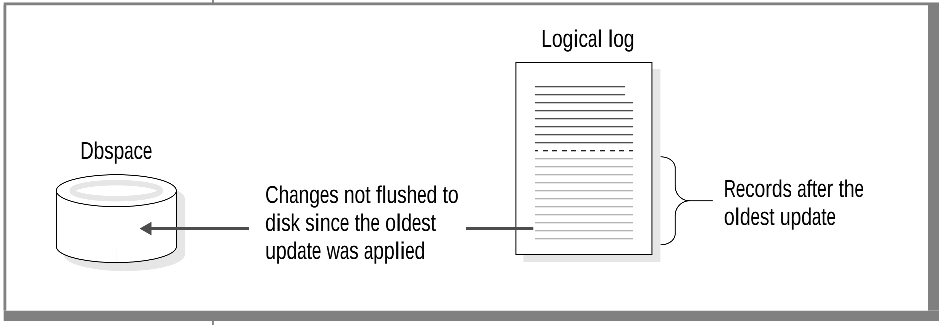

Logical log
Dbspace
Changes not flushed to disk since the oldest update was applied
Records after the oldest update

Figure 24-8
Applying the Log Records for Fuzzy Operations

Rolling Forward Logical-Log Records

In the third step, the database server processes all logical-log records following the last checkpoint. Fast recovery rolls forward the logical-log records that were written after the most recent checkpoint record. This action reproduces all changes to the databases since the time of the last checkpoint, up to the point at which the uncontrolled shutdown occurred. Figure 24-9 illustrates the roll forward of all records after the fuzzy checkpoint.

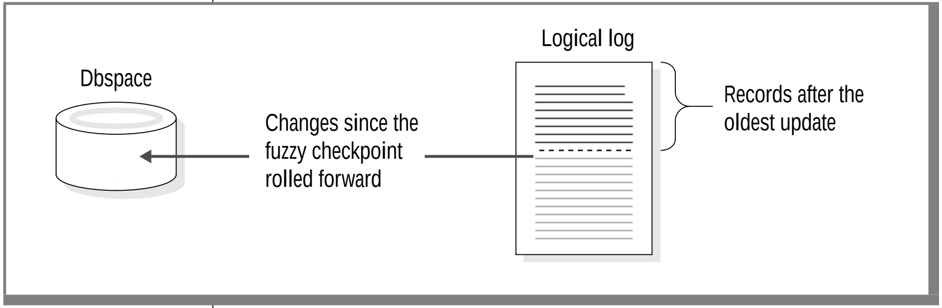

Logical log
Dbspace
Records after the oldest update
Changes since the fuzzy checkpoint rolled forward

Figure 24-9
Rolling Forward the Logical-Log Records Written Since the Most Recent Fuzzy Checkpoint

Rolling Back Incomplete Transactions

The final step in fast recovery rolls back all logical-log records for transactions that were not committed at the time that the system failed. This rollback procedure ensures that all databases are left in a consistent state.

Because one or more transactions possibly spanned several checkpoints without being committed, this rollback procedure might read backward through the logical log past the most recent checkpoint record. All logical-log files that contain records for open transactions are available to the database server because a log file is not freed until all transactions contained within it are closed. Figure 24-10 illustrates the rollback procedure. When fast recovery is complete, the database server goes to quiescent or on-line mode.

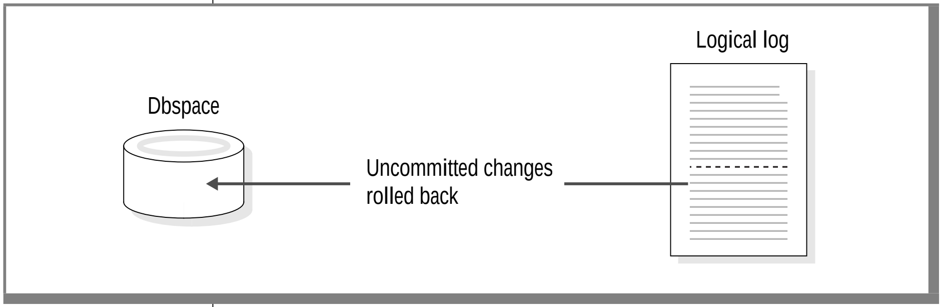


Figure 24-10
Rolling Back All Incomplete Transactions

Fast Recovery of Tables

Figure 24-11 on page 24-24 presents fast-recovery scenarios for the six table types available with Extended Parallel Server. For more information about the table types, see "Table Types" on page 15-25.

Important: *To ensure data integrity when you use raw, static, or operational tables, force a checkpoint or perform a level-0 backup after you modify or load the table.*

Figure 24-11
Fast Recovery of Tables

Table Type	**Fast-Recovery Behavior**
Standard	Fast recovery is successful. All committed log records are rolled forward, and all incomplete transactions are rolled back.
Temp or Scratch	Temp and scratch tables are not recoverable because they are dropped when the database server restarts. All changes made to these tables are lost, and the space can be reused.
Raw	If a checkpoint completed since the raw table was modified last, all the data is recoverable. Updates and deletions that occurred after the last checkpoint are lost. The raw table looks as it did at the last checkpoint. If inserts occurred after the last checkpoint, the raw table might contain empty rows or be corrupted. If a light append was in progress at the time of the failure, all appended records are lost on reboot.
Static	If you alter a raw table to static, you can recover it only if it was not updated since the previous checkpoint. If the raw table was updated before it was altered, it might contain empty rows or be corrupted. If you alter an operational table to static, you can recover it only if light appends did not occur. If light appends occurred, they are lost on reboot. If you alter a standard table to static, you can recover it successfully.
Operational	All inserts, deletions, and updates are successfully recovered. All committed log records are rolled forward and incomplete transactions are rolled back. If a light append was in progress at the time of the failure, all appended records are lost on reboot.

Monitoring Checkpoint Information

Monitor checkpoint activity to determine basic checkpoint information. This information includes the number of times that threads had to wait for the checkpoint to complete. This information is useful for determining if the checkpoint interval is appropriate. For information on tuning the checkpoint interval, see your *Performance Guide*.

Using onstat Options

You can use the following **onstat** options to obtain checkpoint information:

- **-m**
- **-p**

Execute **onstat -m** to view the last 20 entries in the message log. If a check-point record does not appear in the last 20 entries, read the message log directly with a text editor. The database server writes individual checkpoint records to the log when the checkpoint ends. If a checkpoint check occurs, but the database server has no pages to write to disk, the database server does not write any records to the message log.

Execute **onstat -p** to obtain these checkpoint statistics:

- Number of checkpoints that occurred since the database server was brought on-line (**numckpts**)
- Number of times that a user thread waits for a checkpoint to finish (**ckpwaits**)

  The database server prevents a user thread from entering a *critical section* during a checkpoint.

Using SMI Tables

The **sysprofile** table provides the same checkpoint statistics that are available from the **onstat -p** option.

The **sysprofile** table contains two columns, **name** and **value**. The **name** column contains the statistic name, and the **value** column contains the statistic value. These rows contain the following checkpoint information.

Checkpoint Statistic	Description
numckpts	Number of checkpoints that have occurred since the database server was brought on-line
ckptwaits	Number of times that threads waited for a checkpoint to finish to enter a *critical section* during a checkpoint

Fault Tolerance

Section VI

Chapter

Mirroring

25

In This Chapter

The first part of this chapter answers the following basic questions about the database server mirroring feature:

- What are the benefits of mirroring?
- What are the costs of mirroring?
- What happens if you do not mirror?
- What should you mirror?
- What mirroring alternatives exist?

The second part of the chapter discusses the actual mirroring process. The following aspects of the process are discussed:

- What happens when you create a mirrored chunk?
- What are the mirror status flags?
- What is recovery?
- What happens during processing?
- What happens if you stop mirroring?
- What is the structure of a mirrored chunk?

For instructions on how to perform mirroring tasks, refer to Chapter 26, “Using Mirroring.”

Mirroring

Mirroring is a strategy that pairs a *primary* chunk of one defined dbspace with an equal-sized *mirrored chunk.*

Every write to the primary chunk is automatically accompanied by an identical write to the mirrored chunk. This concept is illustrated in Figure 25-1. If a failure occurs on the primary chunk, mirroring enables you to read from and write to the mirrored chunk until you can recover the primary chunk, all without interrupting user access to data.

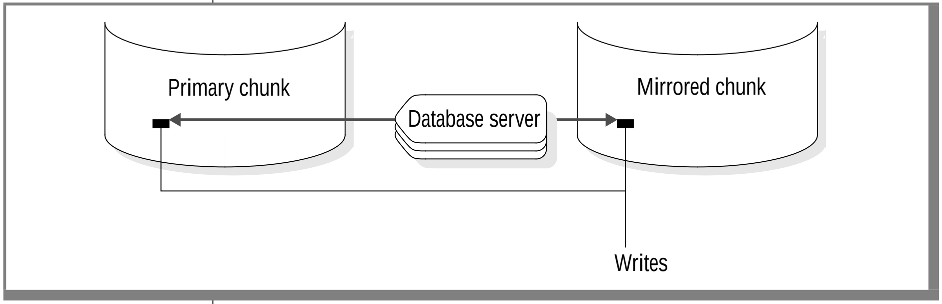


Figure 25-1
Writing Data to Both the Primary Chunk and the Mirrored Chunk

Mirroring is not supported on disks that are managed over a network. The same database server instance must manage all the chunks of a mirrored set.

Benefits of Mirroring

If a media failure occurs, mirroring provides the database server administrator with a means of recovering data without having to take the database server off-line. This feature results in greater reliability and less system downtime. Furthermore, applications can continue to read from and write to a database whose primary chunks are on the affected media, provided that the chunks that mirror this data are located on separate media.

Any critical database should be located in a mirrored dbspace. Above all, the root dbspace, which contains the database server reserved pages, should be mirrored.

Costs of Mirroring

Disk-space costs as well as performance costs are associated with mirroring. The disk-space cost is due to the additional space required for storing the mirror data. The performance cost results from having to perform writes to both the primary and mirrored chunks. The use of multiple virtual processors for disk writes reduces this performance cost. The use of *split reads*, whereby the database server reads data from either the primary chunk or the mirrored chunk, depending on the location of the data within the chunk, actually causes performance to improve for read-only data. For more information on how the database server performs reads and writes for mirrored chunks, see "Actions During Processing" on page 25-9.

Consequences of Not Mirroring

If you do not mirror your dbspaces, the frequency with which you have to restore from a storage-space backup after a media failure increases.

When a mirrored chunk suffers a media failure, the database server reads exclusively from the chunk that is still on-line until you bring the down chunk back on-line. On the other hand, when an *unmirrored* chunk goes down, the database server cannot access the data stored on that chunk. If the chunk contains logical-log files, the physical log, or the root dbspace, the database server goes off-line immediately. If the chunk does not contain logical-log files, the physical log, or the root dbspace, the database server can continue to operate, but threads cannot read from or write to the down chunk. Unmirrored chunks that go down must be restored by recovering the dbspace from a backup.

Data to Mirror

Ideally, you should mirror all of your data. If disk space is an issue, however, you might not be able to do so. In this case, select certain critical chunks to mirror.

Critical chunks always include the chunks that are part of the root dbspace, the chunk that stores the logical-log files, and the chunk that stores the physical logs. If any one of these critical chunks fail, the database server goes off-line immediately.

If some chunks hold data that is critical to your business, give these chunks high priority for mirroring.

Also give priority for mirroring to other chunks that store frequently used data. This action ensures that the activities of many users are not halted if one widely used chunk goes down.

Alternatives to Mirroring

Mirroring, as discussed in this manual, is a database server feature. Your operating system or hardware might provide alternative mirroring solutions.

If you are considering a mirroring feature provided by your operating system instead of database server mirroring, compare the implementation of both features before you decide which to use. The slowest step in the mirroring process is the actual writing of data to disk. The database server strategy of performing writes to mirrored chunks in parallel helps to reduce the time required for this step. (See "Disk Writes to Mirrored Chunks" on page 25-9.) In addition, database server mirroring uses split reads to improve read performance. (See "Disk Reads from Mirrored Chunks" on page 25-9.) Operating-system mirroring features that do not use parallel mirror writes and split reads might provide inferior performance.

Nothing prevents you from running database server mirroring and operating-system mirroring at the same time. They run independently of each other. In some cases, you might decide to use both the database server mirroring and the mirroring feature provided by your operating system. For example, you might have both database server data and other data on a single disk drive. You could use the operating-system mirroring to mirror the other data and database server mirroring to mirror the database server data.

Logical Volume Managers

Logical volume managers are an alternative mirroring solution. Some operating-system vendors provide this type of utility to have multiple disks appear as one file system. Saving data to more than two disks gives you added protection from media failure, but the additional writes have a performance cost.

Hardware Mirroring

Another solution is to use hardware mirroring such as RAID (redundant array of inexpensive disks). An advantage of this type of hardware mirroring is that it requires less disk space than database server mirroring does to store the same amount of data in a manner resilient to media failure. The disadvantage is that it is slower than database server mirroring for write operations.

Mirroring Process

This section describes the mirroring process in greater detail. For instructions on how to perform mirroring operations such as creating mirrored chunks, starting mirroring, changing the status of mirrored chunks, and so on, refer to Chapter 26, "Using Mirroring."

Creation of a Mirrored Chunk

When you specify a mirrored chunk, the database server copies all the data from the primary chunk to the mirrored chunk. This copy process is known as *recovery*. Mirroring begins as soon as recovery is complete.

The recovery procedure that marks the beginning of mirroring is delayed if you start to mirror chunks within a dbspace that contains a logical-log file. Mirroring for dbspaces that contain a logical-log file does not begin until you create a level-0 backup of the root dbspace. The delay ensures that the database server can use the mirrored logical-log files if the primary chunk that contains these logical-log files becomes unavailable during a dbspace restore. The level-0 backup copies the updated database server configuration information, including information about the new mirrored chunk, from the root dbspace reserved pages to the backup. If you perform a data restore, the updated configuration information at the beginning of the backup directs the database server to look for the mirrored copies of the logical-log files if the primary chunk becomes unavailable. If this new storage-space backup information does not exist, the database server is unable to take advantage of the mirrored log files.

For similar reasons, you cannot mirror a dbspace that contains a logical-log file while a dbspace backup is being created. The new information that must appear in the first block of the dbspace backup tape cannot be copied there once the backup has begun.

For more information on creating mirrored chunks, refer to Chapter 26, "Using Mirroring."

Mirror Status Flags

Dbspaces have status flags that indicate whether it is mirrored, unmirrored, or mirrored.

You must perform a level-0 backup of the root dbspace before mirroring starts.

Chunks have status flags that indicate the following information:

- Whether the chunk is a primary or mirrored chunk
- Whether the chunk is currently on-line, down, a new mirrored chunk that requires a level-0 backup of the root dbspace, or being recovered

For descriptions of these chunk status flags, refer to the description of the **onstat -d** option in the utilities chapter of the *Administrator's Reference*. For information on how to display these status flags, refer to "Monitoring Disk Usage" on page 16-34.

Recovery

When the database server recovers a mirrored chunk, it performs the same recovery procedure that it uses when mirroring begins. The mirror-recovery process consists of copying the data from the existing on-line chunk onto the new, repaired chunk until the two are considered identical.

When you initiate recovery, the database server puts the down chunk in recovery mode and copies the information from the on-line chunk to the recovery chunk. When the recovery is complete, the chunk automatically receives on-line status. You perform the same steps whether you are recovering the primary chunk of a mirrored pair or recovering the mirrored chunk.

Tip: *You can still use the on-line chunk while the recovery process is occurring. If data is written to a page that has already been copied to the recovery chunk, the database server updates the corresponding page on the recovery chunk before it continues with the recovery process.*

For information on how to recover a down chunk, refer to the information on recovering a mirrored chunk on page 26-10.

Actions During Processing

This section discusses some of the details of disk I/O for mirrored chunks and how the database server handles media failure for these chunks.

Disk Writes to Mirrored Chunks

During database server processing, the database server performs mirroring by executing two writes for each modification: one to the primary chunk and one to the mirrored chunk. Virtual processors of the AIO class perform the actual disk I/O. For more information, refer to "Asynchronous I/O" on page 11-24.

The requesting thread submits the two write requests (one for the primary chunk and one for the mirrored chunk) asynchronously. That is, if two AIO virtual processors are idle, they can perform the two disk writes in parallel. In the meantime, the requesting thread can perform any additional processing that does not depend on the result of the mirror I/O.

Disk Reads from Mirrored Chunks

The database server uses mirroring to improve read performance because two versions of the data reside on separate disks. A data page is read from either the primary chunk or the mirrored chunk, depending on which half of the chunk includes the address of the data page. This feature is called a *split read*. Split reads improve performance by reducing the disk-seek time. Disk-seek time is reduced because the maximum distance over which the disk head must travel is reduced by half. Figure 25-2 on page 25-10 illustrates a split read.

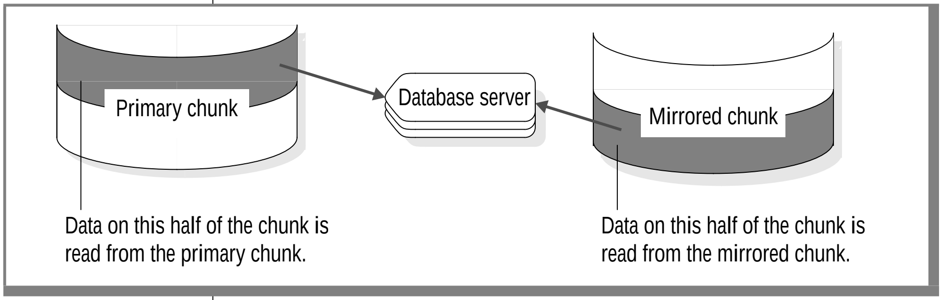

Primary chunk
Database server
Mirrored chunk
Data on this half of the chunk is read from the primary chunk.
Data on this half of the chunk is read from the mirrored chunk.

Figure 25-2
Split Read Reducing the Maximum Distance Over Which the Disk Head Must Travel

Detection of Media Failures

The database server checks the return code when it first opens a chunk and after any read or write. Whenever the database server detects that a primary (or mirror) chunk device has failed, it sets the chunk-status flag to down (D). For information on chunk-status flags, refer to "Mirror Status Flags" on page 25-8.

If the database server detects that a primary (or mirror) chunk device has failed, reads and writes continue for the one chunk that remains on-line. This statement is true even if the administrator intentionally brings down one of the chunks.

Once the administrator recovers the down chunk and returns it to on-line status, reads are again split between the primary and mirrored chunks, and writes are made to both chunks.

Chunk Recovery

The database server uses asynchronous I/O to minimize the time required for recovering a chunk. The read from the chunk that is on-line can overlap with the write to the down chunk, instead of the two processes occurring serially. That is, the thread that performs the read does not have to wait until the thread that performs the write has finished before it reads more data.

Result of Stopping Mirroring

When you end mirroring, the database server immediately frees the mirrored chunks and makes the space available for reallocation. The action of ending mirroring takes only a few seconds.

Create a level-0 backup of the root dbspace after you end mirroring to ensure that the reserved pages with the updated mirror-chunk information are copied to the backup. This action prevents the restore procedure from assuming that mirrored data is still available.

Structure of a Mirrored Chunk

The mirrored chunk contains the same control structures as the primary chunk, so mirrors of dbspace chunks contain dbspace overhead pages.

For information on these structures, refer to the section on the structure of a mirrored chunk in the disk structures and storage chapter of the *Administrator's Reference*.

A display of disk-space use, provided by one of the methods discussed under "Monitoring Chunks" on page 16-34, always indicates that the mirrored chunk is full, even if the primary chunk has free space. The *full* mirrored chunk indicates that none of the space in the chunk is available for use other than as a mirror of the primary chunk. The status remains full for as long as both primary chunk and mirrored chunk are on-line.

If the primary chunk goes down, and the mirrored chunk becomes the primary chunk, disk-space allocation reports then accurately describe the fullness of the new primary chunk.

Chapter

Using Mirroring

26

In This Chapter

This chapter describes the various mirroring tasks that are required to use the database server mirroring feature. It provides an overview of the steps required for mirroring data. Then it describes the following tasks:

- Enabling mirroring
- Allocating disk space for mirrored chunks
- Starting mirroring (creating mirrored chunks)
- Adding chunks to mirrored dbspaces
- Changing the mirror status of chunks
- Modifying mirroring on all coservers
- Relinking mirrored chunks after a disk failure
- Ending mirroring

Steps Required for Mirroring Data

To start mirroring data on a database server that is not running with the mirroring function enabled, you must perform the following steps:

1. Take the database server off-line and enable mirroring. See "Enabling Mirroring" on page 26-4.
2. Bring the database server back on-line.
3. Allocate disk space for the mirrored chunks. You can allocate this disk space at any time, as long as the disk space is available when you specify mirrored chunks in the next step. See "Allocating Disk Space for Mirrored Data" on page 26-5.

4. Choose the dbspace that you want to mirror, and create mirrored chunks by specifying a mirror-chunk pathname and offset for each primary chunk in that storage space. The mirroring process starts after you perform this step. Repeat this step for all the storage spaces that you want to mirror. See "Using Mirroring" on page 26-6.

Enabling Mirroring

When you enable mirroring, you invoke the database server functionality required for mirroring tasks. However, when you enable mirroring, you do not initiate the mirroring process. Mirroring does not actually start until you create mirrored chunks for a dbspace, blobspace, or sbspace. See "Using Mirroring" on page 26-6.

Enable mirroring when you initialize the database server if you plan to create a mirror for the root dbspace as part of initialization; otherwise, leave mirroring disabled. If you later decide to mirror a storage space, you can change the value of the MIRROR configuration parameter.

Changing the MIRROR Parameter with ONCONFIG

To enable mirroring for the database server, you must set the MIRROR parameter in ONCONFIG to 1. The default value of MIRROR is 0, indicating that mirroring is disabled.

To change the value of MIRROR, you can edit the ONCONFIG file with a text editor while the database server is in on-line mode. After you change the ONCONFIG file, reinitialize shared memory (take the database server off-line and then to quiescent mode) for the change to take effect.

Allocating Disk Space for Mirrored Data

Before you can create a mirrored chunk, you must allocate disk space for this purpose. You can allocate either raw disk space or cooked file space for mirrored chunks. For a discussion of allocating disk space, refer to “Allocating Disk Space” on page 16-6.

Always allocate disk space for a mirrored chunk on a different disk than the corresponding primary chunk with, ideally, a different controller. This setup allows you to access the mirrored chunk if the disk on which the primary chunk is located goes down, or vice versa.

Linking Chunks

Use the UNIX link (**ln**) command to link the actual files or raw devices of the mirrored chunks to mirror pathnames. If a disk failure occurs, you can link a new file or raw device to the pathname, eliminating the need to physically replace the disk that failed before the chunk is brought back on-line.

Relinking a Chunk to a Device After a Disk Failure

On UNIX, if the disk on which the actual mirror file or raw device is located goes down, you can relink the chunk to a file or raw device on a different disk. This action allows you to recover the mirrored chunk before the disk that failed is brought back on-line. Typical UNIX commands that you can use for relinking are shown in the following examples.

The original setup consists of a primary root chunk and a mirror root chunk, which are linked to the actual raw disk devices, as follows:

```
% ln -lg
lrwxrwxrwx 1 informix 10 May 3 13:38 /dev/root@->/dev/rxy0h
lrwxrwxrwx 1 informix 10 May 3 13:40 /dev/mirror_root@->/dev/rsd2b
```

Assume that the disk on which the raw device **/dev/rsd2b** resides has gone down. You can use the **rm** command to remove the corresponding symbolic link, as follows:

```
% rm /dev/mirror_root
```

Now you can relink the mirrored chunk pathname to a raw disk device, on a disk that is running, and proceed to recover the chunk, as follows:

```
% ln -s /dev/rab0a /dev/mirror_root
```

Using Mirroring

Mirroring starts when you create a mirrored chunk for each primary chunk in a dbspace.

This action consists of specifying disk space that you have already allocated, either raw disk space or a cooked file, for each mirrored chunk.

When you create a mirrored chunk, the database server performs the *recovery* process, copying data from the primary chunk to the mirrored chunk. When this process is complete, the database server begins mirroring data. If the primary chunk contains logical-log files, the database server does not perform the recovery process immediately after you create the mirrored chunk but waits until you perform a level-0 backup. For an explanation of this behavior see “Creation of a Mirrored Chunk” on page 25-7.

You must always start mirroring for an entire dbspace. The database server does not permit you to select particular chunks in a dbspace to mirror.

When you select a space to mirror, you must create mirrored chunks for every chunk within the space.

You start mirroring a storage space when you perform the following operations:

- Create a mirrored root dbspace during system initialization
- Change the status of a dbspace from unmirrored to mirrored
- Create a mirrored dbspace

Each of these operations requires you to create mirrored chunks for the existing chunks in the storage space.

Mirroring the Root Dbspace During Initialization

If you enable mirroring when you initialize the database server, you can also specify a mirror pathname and offset for the root chunk. The database server creates the mirrored chunk when the server is initialized. However, because the root chunk contains logical-log files, mirroring does not actually start until you perform a level-0 backup.

To specify the root mirror pathname and offset, set the values of MIRRORPATH and MIRROROFFSET in the ONCONFIG file before you bring up the database server.

If you do not provide a mirror pathname and offset, but you do want to start mirroring the root dbspace, you must change the mirroring status of the root dbspace once the database server is initialized.

Changing the Mirror Status

You can make the following two changes to the status of a mirrored chunk:

- Change a mirrored chunk from on-line to down
- Change a mirrored chunk from down to recovery

You can take down or restore a chunk only if it is part of a mirrored pair. You can take down either the primary chunk or the mirrored chunk, as long as the other chunk in the pair is on-line.

For information on how to determine the status of a chunk, refer to "Monitoring Disk Usage" on page 16-34.

Taking Down a Mirrored Chunk

When a mirrored chunk is *down*, the database server cannot write to it or read from it. You might take down a mirrored chunk to relink the chunk to a different device. (See "Relinking a Chunk to a Device After a Disk Failure" on page 26-5.)

Taking down a chunk is not the same as ending mirroring. You end mirroring for a complete dbspace, which causes the database server to drop all the mirrored chunks for that dbspace.

Ending Mirroring

When you end mirroring for a dbspace, the database server immediately releases the mirrored chunks of that dbspace. These chunks are immediately available for reassignment to other dbspaces.

Only users **informix** and **root** can initiate this action.You cannot end mirroring if any of the primary chunks in the dbspace are down. The system can be in on-line mode when you end mirroring.

Ending Mirroring with onutil

When you end mirroring for a dbspace, the database server immediately releases the mirrored chunks of that space.

You can end mirroring with the **onutil** utility. For example, to end mirroring for the root dbspace, enter the following command:

```
% onutil
1> ALTER DBSPACE rootdbs STOP MIRRORING
```

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

Managing Mirroring in Extended Parallel Server

Use the **onutil** ALTER DBSPACE command to manage mirroring.

Starting Mirroring for Unmirrored Dbspaces

To start mirroring for a dbspace, use the **onutil** ALTER DBSPACE command with the START MIRRORING clause.

The following example shows how to start mirroring for a dbspace:

```
% onutil
1> alter dbspace dbsp1 start mirroring
2> chunk "/dev/chk1" offset 1024
3> mirror "/dev/mirror1" offset 1024 ;
Starting Mirroring for New Dbspaces and Dbslices
```

Starting Mirroring for New Dbspaces and Dbslices

To start mirroring when you create a new dbslice, use the **onutil** CREATE DBSLICE command with the MIRROR clause. The following example shows how to create a dbslice with mirroring for all of its dbspaces:

```
% onutil
1> create dbslice acctg_sl
2> from cogroup acctg_cogroup
3> chunk "/dev/dbsl_acctg.%r(1..2)"
4> offset 1024 size 1024
5> mirror "/dev/mirr_dbsl_acctg.%r(1..2)";
```

Adding Mirrored Chunks to a Dbspace

If you add a chunk to a dbspace that is mirrored, you must also add a corresponding mirrored chunk. You can use the **onutil** ALTER DBSPACE command to add chunks to a dbspace and its mirror, as the following example shows:

```
% onutil
1> alter dbspace dbsp1 add chunk
2> chunk "/dev/chunk2" offset 1024 size 1 gbytes
3> mirror "/dev/mirror2" offset 1024 ;
```

Taking Down a Mirrored Chunk

When a mirrored chunk is *down*, the database server cannot write to it or read from it. You might take down a mirrored chunk to relink the chunk to a different device. (See "Relinking a Chunk to a Device After a Disk Failure" on page 26-5.) Taking down a chunk is not the same as ending mirroring. You end mirroring for a complete dbspace, which causes the database server to drop all the mirrored chunks for that dbspace. Use the **onutil** ALTER DBSPACE command with the OFFLINE clause to take down a chunk.

```
% onutil
1> alter dbspace dbsp1 (offline)
2> chunk "/dev/chunk2" offset 1024;
```

Recovering a Mirrored Chunk

You recover a down chunk to begin mirroring the data in the chunk that is on-line. Use the **onutil** ALTER DBSPACE command with the ONLINE clause to recover a down chunk.

```
% onutil
1> alter dbspace dbsp1 (online)
2> chunk "/dev/chunk2" offset 1024;
```

Modifying Mirroring of All Root Dbspaces

You can change mirroring for all root dbspaces on all coservers in your database server.

1. Shut down Extended Parallel Server (bring down all coservers) with the following command:

   ```
   xctl onmode -ky
   ```

2. Add the MIRROR, MIRRORPATH, and MIRROROFFSET configuration parameters in the global parameter section of your ONCONFIG file.

   The following is an excerpt from the global section of a sample ONCONFIG file to turn on mirroring for all root dbspaces in Extended Parallel Server:

   ```
   DBSERVERNAME  xps

   ROOTSLICE     rootdbs
   ROOTPATH      /work/dbspaces/rootdbs_%c
   ROOTOFFSET    0
   ROOTSIZE      40000

   MIRROR        1   # 1 = yes
   MIRRORPATH    /work/dbspaces/mirror_%c
   MIRROROFFSET  0
   ```

3. Bring up the database server in either on-line or quiescent mode with the following command:

   ```
   xctl -b -X= oninit -X=
   ```

Important: *Turn mirroring on or off for all coservers.*

Ending Mirroring for a Dbspace

When you end mirroring for a dbspace, the database server releases the mirrored chunks. These chunks are immediately available for reassignment to other dbspaces. You cannot end mirroring if any of the primary chunks in the dbspace are down. The database server can be in on-line mode when you end mirroring. Use the **onutil** ALTER DBSPACE command with the STOP MIRRORING clause to end mirroring.

```
% onutil
1> alter dbspace dbsp1 stop mirroring
```

Chapter

Consistency Checking

27

In This Chapter

Informix database servers are designed to detect database server malfunctions or problems caused by hardware or operating-system errors. It detects problems by performing *assertions* in many of its critical functions. An *assertion* is a consistency check that verifies that the contents of a page, structure, or other entity match what would otherwise be assumed.

When one of these checks finds that the contents are not what they should be, the database server reports an *assertion failure* and writes text that describes the check that failed in the database server message log. The database server also collects further diagnostics information in a separate file that might be useful to Informix Technical Support staff.

This chapter provides an overview of consistency-checking measures and ways of handling inconsistencies. It covers the following topics:

- Performing periodic consistency checking
- Dealing with data corruption
- Collecting advanced diagnostic information

Performing Periodic Consistency Checking

To gain the maximum benefit from consistency checking and to ensure the integrity of dbspace backups, Informix recommends that you periodically take the following actions:

- Verify that all data and the database server overhead information is consistent.
- Check the message log for assertion failures while you verify consistency.
- Create a level-0 dbspace backup after you verify consistency.

The following sections describe each of these actions.

Verifying Consistency

Because of the time needed for this check and the possible contention that the check can cause, schedule this check for times when activity is at its lowest. Informix recommends that you perform this check just before you create a level-0 dbspace backup.

Run the **onutil** commands shown in Figure 27-1 as part of the consistency check.

Figure 27-1
Checking Data Consistency

Type of Validation	Command
Data	**onutil** CHECK DATA IN DATABASE *dbname*
Extents	**onutil** CHECK SPACE
Indexes	**onutil** CHECK INDEX WITH DATA DATABASE *dbname* or **onutil** CHECK ALLOCATION INFO
Logical logs	**onutil** CHECK LOGS
Reserved pages	**onutil** CHECK RESERVED
System catalog tables	**onutil** CHECK CATALOGS

You can run each of these commands while the database server is in on-line mode. For information about how each command locks objects as it checks them and which users can perform validations, see **onutil** in the *Administrator's Reference*.

In most cases, if one or more of these validation procedures detects an error, the solution is to restore the database from a dbspace backup. However, the source of the error might also be your hardware or operating system.

Validating Data Pages

To validate data pages, use the **onutil** CHECK DATA command.

If data-page validation detects errors, try to unload the data from the specified table, drop the table, re-create the table, and reload the data. For information about loading and unloading data, see the *Informix Migration Guide*. If this procedure does not succeed, perform a data restore from a storage-space backup.

Validating Extents

To validate extents in every database, use the **onutil** CHECK SPACE command.

Extents must not overlap. If this command detects errors, perform a data restore from a storage-space backup.

Validating Indexes

To validate indexes on each of the tables in the database, use the **onutil** CHECK INDEX WITH DATA command.

If this command detects errors, drop and re-create the affected index.

Validating Logical Logs

To validate logical logs, use the **onutil** CHECK LOGS command.

Validating Reserved Pages

To validate reserved pages, use the **onutil** CHECK RESERVED command.

Reserved pages are pages that reside at the beginning of the initial chunk of the root dbspace. These pages contain the primary database server overhead information. If this command detects errors, perform a data restore from storage-space backup.

This command might provide warnings. In most cases, these warnings call your attention to situations of which you are already aware.

Validating System Catalog Tables

To validate system catalog tables, use the **onutil** CHECK CATALOGS command.

Each database contains its own system catalog, which contains information about the database tables, columns, indexes, views, constraints, stored procedures, and privileges.

If a warning appears when validation completes, its only purpose is to alert you that no records of a specific type were found. These warnings do not indicate any problem with your data, your system catalog, or even your database design. For example, the following warning might appear if you validate system catalog tables for a database that has no synonyms defined for any table:

```
WARNING: No syssyntable records found.
```

This message indicates only that no synonym exists for any table; that is, the system catalog contains no records in the table **syssyntable**.

However, if you receive an error message when you validate system catalog tables, the situation is quite different. Contact Informix Technical Support immediately.

Monitoring for Data Inconsistency

If the consistency-checking code detects an inconsistency during database server operation, an assertion failure is reported to the database server message log. (See the message-log chapter in the *Administrator's Reference*.)

Assertion Failures in the Message Log and Dump Files

Figure 27-2 shows the form that assertion failures take in the message log.

```
Assert Failed: Short description of what failed
    Who: Description of user/session/thread running at the time
    Result: State of the affected database server entity
    Action: What action the database server administrator should take
    See Also: file(s) containing additional diagnostics
```

Figure 27-2
Form of Assertion Failures in the Message Log

The `See Also`: line contains one or more of the following filenames:

- **af**.*xxx*
- **shmem**.*xxx*
- / ***pathname*** / **core**

In all cases, ***xxx*** is a hexadecimal number common to all files associated with the assertion failures of a single thread. The files **af**.*xxx*, **shmem**.*xxx*, and **gcore**.*xxx* are in the directory that the ONCONFIG parameter DUMPDIR specifies.

The file **af**.*xxx* contains a copy of the assertion-failure message that was sent to the message log, as well as the contents of the current, relevant structures and data buffers.

The file **shmem**.*xxx* contains a complete copy of the database server shared memory at the time of the assertion failure, but only if the ONCONFIG parameter DUMPSHMEM is set to 1.

The file **gcore**.*xxx* contains a core dump of the database server virtual process on which the thread was running at the time, but only if the ONCONFIG parameter DUMPGCORE is set to 1 and your operating system supports the **gcore** utility. The **core** file contains a core dump of the database server virtual process on which the thread was running at the time, but only if the ONCONFIG parameter DUMPCORE is set to 1. The pathname for the **core** file is the directory from which the database server was last invoked.

Validating Table and Tablespace Data

To validate table and tablespace data, use the **onutil** CHECK TABLE DATA command on the database or table.

Most of the general assertion-failure messages are followed by additional information that usually includes the tblspace where the error was detected. If this check verifies the inconsistency, unload the data from the table, drop the table, re-create the table, and reload the data. Otherwise, no other action is needed.

In many cases, the database server stops immediately when an assertion fails. However, when failures appear to be specific to a table or smaller entity, the database server continues to run.

When an assertion fails because of inconsistencies on a data page that the database server accesses on behalf of a user, an error is also sent to the application process. The SQL error depends on the operation in progress. However, the ISAM error is almost always be either -105 or -172, as follows:

```
-105 ISAM error: bad isam file format
-172 ISAM error: Unexpected internal error
```

For additional details about the objectives and contents of messages, see the chapter on message-log messages in the *Administrator's Reference*.

Retaining Consistent Level-0 Backups

After you perform the checks described in "Verifying Consistency" on page 27-4 without errors, create a level-0 backup. Retain this storage-space backup and all subsequent logical-log backup tapes until you complete the next consistency check. Informix recommends that you perform the consistency checks before every level-0 backup. If you do not, then at minimum keep all the tapes necessary to recover from the storage-space backup that was created immediately after the database server was verified to be consistent.

Dealing with Corruption

This section describes some of the symptoms of database server system corruption and actions that the database server or you, as administrator, can take to resolve the problems. Corruption in a database can occur as a consequence of hardware or operating-system problems, or from some unknown database server problems. Corruption can affect either data or database server overhead information.

Symptoms of Corruption

The database server alerts the user and administrator to possible corruption in the following ways:

- Error messages reported to the application state that pages, tables, or databases cannot be found. One of the following errors is always returned to the application if an operation has failed because of an inconsistency in the underlying data or overhead information:

  ```
  -105 ISAM error: bad isam file format
  -172 ISAM error: Unexpected internal error
  ```

- Assertion-failure reports are written to the database server message log. They always indicate files that contain additional diagnostic information that can help you determine the source of the problem. See "Monitoring for Data Inconsistency" on page 27-6.
- The **onutil** utility returns errors.

Fixing Index Corruption

At the first indication of corruption, run the **onutil** CHECK INDEX WITH DATA command to determine if corruption exists in the index.

If you run **onutil** CHECK INDEX in quiescent mode, and corruption is detected, you are prompted to confirm whether the utility should attempt to repair the corruption. The **onutil repair** keyword automatically repairs corruption if it can, without prompting you for confirmation.

If your utility reports bad key information in an index, drop the index and re-create it. If your utility cannot find or access the table or database, perform the checks described in "Verifying Consistency" on page 27-4.

I/O Errors on a Chunk

If an I/O error occurs during the database server operation, the status of the chunk on which the error occurred changes to down. If a chunk is down, the **onstat -d** display shows the chunk status as `PD-` for a primary chunk and `MD-` for a mirrored chunk.

Figure 27-3 shows an example in which chunk 2 is down.

```
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
40c980   1        1        1        1        N        informix rootdbs
40c9c4   2        1        2        1        N        informix fstdbs
 2 active, 8192 maximum

Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
40c224   1   1   0        20000    14001             PO-   /home/server/root_chunk
40c2bc   2   2   0        2000     1659              PD-   /home/server/fst_chunk
 2 active, 8192 maximum
```

Figure 27-3
onstat -d Output

Additionally, the message log lists a message with the location of the error and a suggested solution. The listed solution is a possible fix but does not always correct the problem.

If the down chunk is mirrored, the database server continues to operate using the mirrored chunk. Use operating-system utilities to determine what is wrong with the down chunk and correct the problem. You must then direct the database server to restore mirrored chunk data.

For information about recovering a mirrored chunk, refer to "Recovering a Mirrored Chunk" on page 26-10.

If the down chunk is not mirrored and contains logical-log files, the physical log, or the root dbspace, the database server immediately initiates an abort. Otherwise, the database server can continue to operate but cannot write to or read from the down chunk or any other chunks in the dbspace of that chunk. You must take steps to determine why the I/O error occurred, correct the problem, and restore the dbspace from a backup.

If you take the database server to off-line mode when a chunk is marked as down (`D`), you can reinitialize the database server, provided that the chunk marked as down does not contain critical data (logical-log files, the physical log, or the root dbspace).

Collecting Diagnostic Information

Several ONCONFIG parameters affect the way in which the database server collects diagnostic information. Because an assertion failure is generally an indication of an unforeseen problem, notify Informix Technical Support whenever one occurs. The diagnostic information collected is intended for the use of Informix technical staff. The contents and use of **af**.*xxx* files and shared core are not further documented.

To determine the cause of the problem that triggered the assertion failure, it is critically important that you not destroy diagnostic information until Informix Technical Support indicates that you can do so. Send a fax or email with the **af**.*xxx* file to Informix Technical Support. This file often contains information that they need to resolve the problem.

Several ONCONFIG parameters direct the database server to preserve diagnostic information whenever an assertion failure is detected or whenever the database server enters into an abort sequence:

- DUMPDIR
- DUMPSHMEM
- DUMPCNT
- DUMPCORE
- DUMPGCORE

For more information about the configuration parameters, see the *Administrator's Reference*.

You decide whether to set these parameters. Diagnostic output can consume a large amount of disk space. (The exact content depends on the environment variables set and your operating system.) The elements of the output could include a copy of shared memory and a core dump.

Tip: *A core dump is an image of a process in memory at the time that the assertion failed. On some systems, core dumps include a copy of shared memory. Core dumps are useful only if this is the case.*

Database server administrators with disk-space constraints might prefer to write a script that detects the presence of diagnostic output in a specified directory and sends the output to tape. This approach preserves the diagnostic information and minimizes the amount of disk space used.

Index