

# **INFORMIX<sup>®</sup>-Universal Server**

## **Administrator's Supplement**

for Windows NT<sup>™</sup>

Version 9.12  
October 1997  
Part No. 000-3918

Published by INFORMIX® Press

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

Copyright © 1981-1997 by Informix Software, Inc. or their subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

INFORMIX®; INFORMIX®-OnLine Dynamic Server™; DataBlade®

The following are worldwide trademarks of the indicated owners or their subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

Adobe Systems Incorporated: PostScript®  
Apple Computer, Inc.: Apple®; Macintosh™  
IBM: OS/2®  
Microsoft Corporation: Microsoft®; MS-DOS®, Windows NT™  
X/Open Company Ltd.: UNIX®, X/Open®

All other marks or symbols are registered trademarks or trademarks of their respective owners.

Documentation Team: Twila Booth, Diana Chase, Karin Kristenson

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

#### RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

# Table of Contents

## Introduction

About This Manual . . . . .	3
Organization of This Manual . . . . .	3
Types of Users . . . . .	4
Software Dependencies . . . . .	4
Assumptions About Your Locale. . . . .	4
Demonstration Database . . . . .	5
Major Features . . . . .	5
Documentation Conventions . . . . .	6
Typographical Conventions . . . . .	7
Icon Conventions . . . . .	8
Command-Line Conventions . . . . .	9
Additional Documentation . . . . .	11
On-Line Manuals . . . . .	12
Printed Manuals . . . . .	12
Error Message Files . . . . .	12
Documentation Notes and Release Notes. . . . .	13
Compliance with Industry Standards . . . . .	13
Informix Welcomes Your Comments . . . . .	14

## Chapter 1

### Universal Server for Windows NT

Windows NT Threads and Universal Server Threads . . . . .	1-3
Virtual Processors Implemented as Windows NT Threads . . . . .	1-3
Virtual-Processor Classes for Windows NT . . . . .	1-5
Informix-Admin Group . . . . .	1-6
Allocating Disk Space . . . . .	1-7
Universal Server for Windows NT as a Windows NT Service . . . . .	1-8
Starting and Stopping Universal Server for Windows NT . . . . .	1-8
The Windows NT Registry . . . . .	1-9
How Universal Server Uses the Windows NT Registry . . . . .	1-10

The Windows NT Event Logs . . . . .	1-11
The Message Server . . . . .	1-12
Auditing . . . . .	1-12
Event Logs for Auditing . . . . .	1-13
The onaudit Utility . . . . .	1-14
The onshowaudit Utility . . . . .	1-17
The Location of Universal Server Files . . . . .	1-19

## **Chapter 2      Client/Server Communications**

Universal Server and Windows NT Network Domains . . . . .	2-3
Connecting to Universal Server for Windows NT . . . . .	2-4
Connecting Different Types of Clients . . . . .	2-5
Creating a TCP/IP Connection to Universal Server . . . . .	2-6
The sqlhosts Registry . . . . .	2-8
How the sqlhosts Registry Is Created . . . . .	2-8
The Location of the sqlhosts Registry . . . . .	2-11
Changing the sqlhosts Registry . . . . .	2-12

## **Chapter 3      Tables and Fragmentation**

Placement of Tables on Disk . . . . .	3-3
Isolating High-Use Tables . . . . .	3-5
Using Multiple Disks for a Dbspace . . . . .	3-6
Improving Smart-Large-Object Metadata I/O . . . . .	3-6
Spreading Temporary Tables and Sort Files Across Multiple Disks . . . . .	3-7
Backup and Restore Considerations . . . . .	3-8
Improving Performance for Nonfragmented Tables and Table Fragments . . . . .	3-8
Estimating the Size of a Table and Index . . . . .	3-8
Managing Indexes . . . . .	3-29
Managing Extents . . . . .	3-36
Managing Smart Large Objects . . . . .	3-45
Denormalizing the Data Model to Improve Performance . . . . .	3-46
Using Shorter Rows for Faster Queries . . . . .	3-46
Expelling Long Strings . . . . .	3-47
Splitting Wide Tables . . . . .	3-48
Redundant Data . . . . .	3-50
Table-Fragmentation Guidelines . . . . .	3-51
Considering Fragmentation Strategy . . . . .	3-54
Distribution Scheme . . . . .	3-60
Monitoring Fragment Use . . . . .	3-64
Fragmenting Indexes . . . . .	3-64
Improving Fragmentation . . . . .	3-65

## Chapter 4

### Queries and the Query Optimizer

The Query Optimizer . . . . .	4-5
Factors That Affect Query-Plan Selection . . . . .	4-6
Displaying the Query Plan . . . . .	4-14
Time Costs of a Query . . . . .	4-15
Activities in Memory . . . . .	4-15
Time Used for a Sort . . . . .	4-16
The Cost of Reading a Row . . . . .	4-17
The Cost of Sequential Access . . . . .	4-18
The Cost of Nonsequential Access . . . . .	4-19
The Cost of Index Look-Ups . . . . .	4-19
The Cost of In-Place ALTER TABLE . . . . .	4-20
The Cost of Small Tables . . . . .	4-20
The Cost of Data Mismatches . . . . .	4-20
The Cost of GLS Functionality . . . . .	4-22
The Cost of Network Access . . . . .	4-22
The Importance of Table Order . . . . .	4-23
Improving Performance for a Particular Query . . . . .	4-29
Improving Filter Selectivity . . . . .	4-30
Creating Data Distributions . . . . .	4-33
Improving Query Performance with Indexes . . . . .	4-34
Reducing the Effect of Join and Sort Operations . . . . .	4-42
Improving Sequential Scans . . . . .	4-45
Reviewing the Optimization Level . . . . .	4-47
Choosing an Index Type . . . . .	4-47
Defining Indexes for User-Defined Data Types . . . . .	4-48
Choosing Operator Classes for Indexes . . . . .	4-57
What Is an Operator Class? . . . . .	4-57
Built-In B-Tree Operator Class . . . . .	4-59
Determining the Available Operator Classes . . . . .	4-61
Using an Operator Class . . . . .	4-62
Managing PDQ Resource Requirements and Queries . . . . .	4-64
How the Optimizer Structures a PDQ Query . . . . .	4-64
The Memory Grant Manager . . . . .	4-65
Allocating Resources for PDQ Queries . . . . .	4-67
Managing Applications . . . . .	4-74

<b>Chapter 5</b>	<b>Using ontape to Back Up and Restore Data</b>	
	Backing Up and Restoring Universal Server Data . . . . .	5-4
	What Is an Archive? . . . . .	5-4
	What Is a Logical-Log Backup? . . . . .	5-5
	What Is a Universal Server Restore? . . . . .	5-7
	Setting the ontape Configuration Parameters . . . . .	5-9
	Setting ontape Parameters with a Text Editor . . . . .	5-10
	Creating a Level-0 Archive After You Change Parameters for ontape . . . . .	5-10
	Setting the Tape-Device Parameters . . . . .	5-10
	Setting the Tape-Block-Size Parameters . . . . .	5-13
	Setting the Tape-Size Parameters . . . . .	5-13
	Checking ontape Configuration Parameters . . . . .	5-14
	Using ontape . . . . .	5-14
	Syntax . . . . .	5-14
	Creating an Archive . . . . .	5-15
	Backing Up Logical-Log Files . . . . .	5-34
	Restoring Universal Server Data . . . . .	5-43
	Changing Database Logging Status . . . . .	5-55
<b>Appendix A</b>	<b>Summary of Universal Server for Windows NT</b>	
<b>Appendix B</b>	<b>Using OSI TP4</b>	
	<b>Index</b>	

# Introduction

About This Manual . . . . .	3
Organization of This Manual . . . . .	3
Types of Users . . . . .	4
Software Dependencies . . . . .	4
Assumptions About Your Locale . . . . .	4
Demonstration Database . . . . .	5
Major Features . . . . .	5
Documentation Conventions . . . . .	6
Typographical Conventions . . . . .	7
Icon Conventions . . . . .	8
Comment Icons . . . . .	8
Feature and Compliance Icons . . . . .	8
Command-Line Conventions . . . . .	9
Additional Documentation . . . . .	11
On-Line Manuals . . . . .	12
Printed Manuals . . . . .	12
Error Message Files . . . . .	12
Documentation Notes and Release Notes . . . . .	13
Compliance with Industry Standards . . . . .	13
Informix Welcomes Your Comments . . . . .	14





# R

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

---

## About This Manual

This manual is a supplement to the *INFORMIX-Universal Server Administrator's Guide*. It describes how Universal Server for the Windows NT operating system differs from Universal Server for the UNIX operating system.

## Organization of This Manual

This manual includes the following chapters:

- This Introduction provides an overview of the manual and describes the documentation conventions used.
- Chapter 1, “Universal Server for Windows NT,” describes how Universal Server for Windows NT is implemented differently for the Windows NT operating system than it is for the UNIX operating system.
- Chapter 2, “Client/Server Communications,” describes the fundamentals of using the TCP/IP protocol for communication between Universal Server and a client application.
- Chapter 3, “Tables and Fragmentation,” describes how to place and fragment tables to improve performance.
- Chapter 4, “Queries and the Query Optimizer,” describes how queries and the query optimizer work.

- Chapter 5, “Using ontape to Back Up and Restore Data,” describes how to use the **ontape** utility to back up and restore Universal Server data.
- Appendix A, “Summary of Universal Server for Windows NT,” provides a summary of the differences between Universal Server for Windows NT and Universal Server for UNIX.
- Appendix B, “Using OSI TP4,” describes how to implement OSI TP4 for server-to-server communications.

## Types of Users

This manual is written for Universal Server for Windows NT administrators and others who need to reference administrative tasks. This supplement assumes that the Universal Server administrator is familiar with both the UNIX and Windows NT operating systems.

If you have limited experience with relational databases, SQL, or your operating system, refer to *Getting Started with INFORMIX-Universal Server* for a list of introductory texts.

## Software Dependencies

This manual assumes that you are using INFORMIX-Universal Server for Windows NT, Version 9.12, as your database server.

In this manual, all instances of Universal Server refer to INFORMIX-Universal 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 GLS (Global Language Support) locale.

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

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale(s). For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Guide to GLS Functionality*.

## Demonstration Database

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. The sample command files that make up a demonstration application are also included.

The script that you use to install the demonstration database is called **dbaccessdemo7.bat**. For a complete explanation of how to create and populate the demonstration database on your database server, refer to the *DB-Access User Manual*.

---

## Major Features

This section highlights the major features of INFORMIX-Universal Server for Windows NT, Version 9.12:

- Support for extensions to the database server in the form of DataBlade modules and user-defined routines
- Support for the storage of smart large objects (CLOB and BLOB data types) in addition to simple large objects (TEXT and BYTE data types)
- Ability to access virtual tables (extspaces), such as a file with columnar data or a spreadsheet, from within a database
- Support for user-defined access methods

- Ability to designate and start a virtual processor in which to run user-defined routines (VPCLASS configuration parameter)
- New configuration parameters: VPCLASS and SBSPACENAME
- Ability to add new VP classes (**onmode**) dynamically
- Core extensibility and scalability
- Extensions to the following utilities to support Universal Server features: **oncheck**, **oninit**, **onlog**, **onmode**, **onspaces**

The Introduction to each Version 9.1x product manual contains a list of major features for that product. The Introduction to each manual in the Version 9.1x *Informix Guide to SQL* series contains a list of new SQL features.

Major features for Version 9.1x Informix products also appear in release notes.

---

## Documentation Conventions

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

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Command-line conventions

## Typographical Conventions

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

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

Icon	Description
	The <i>warning</i> icon identifies vital instructions, cautions, or critical information.
	The <i>important</i> icon identifies significant information about the feature or operation that is being described.
	The <i>tip</i> icon identifies additional details or shortcuts for the functionality that is being described.

### Feature and Compliance Icons

Feature icons identify paragraphs that contain feature-specific information. Compliance icons identify paragraphs that provide guidelines for complying with a standard.

Icon	Description
	Identifies information that relates to the Informix Global Language Support (GLS) feature.
	Identifies information that is valid only if your database is ANSI compliant.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature-specific information.

## Command-Line Conventions

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

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

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

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
<i>variable</i>	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.

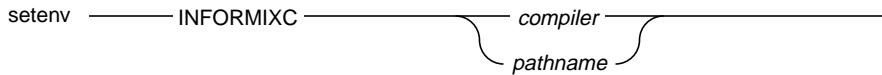
(1 of 2)

Element	Description
.ext	A filename extension, such as .sql or .cob, might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file and a period. The extension might be optional in certain products.
(.,;+*- / )	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
<div>Privileges p. 5-17</div> <div>Privileges</div>	A reference in a box represents a subdiagram on the same page (if no page is supplied) or another page. Imagine that the subdiagram is spliced into the main diagram at this point.
<div>Showing Audit Masks see TFM</div>	A reference to TFM in this manual refers to the <i>INFORMIX-Universal Server Trusted Facility Manual</i> . Imagine that the subdiagram is spliced into the main diagram at this point.
— ALL —	A shaded option is the default action.
→ →	Syntax within a pair of arrows indicates a subdiagram.
—	The vertical line terminates the statement.
-f — OFF — ON —	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
— ' — variable	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items, as in this example.
— /3 — size	A gate ( /3 ) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify size no more than three times within this statement segment.



Figure 1 shows how you read the command-line diagram for setting the **INFORMIXC** environment variable.

**Figure 1**  
Example of a Command-Line Diagram



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

**Important:** This example command-line diagram and the following explanation are meant only to show how to read command-line syntax. The **INFORMIXC** environment variable is set by the installation program.

Figure 1 diagrams the following steps:

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

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

4. Press RETURN to execute the command.

## Additional Documentation

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

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes and release notes



## On-Line Manuals

A 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 either the installation guide for your product or the installation insert that accompanies the documentation CD.

The documentation set that is provided on the CD describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see *Getting Started with INFORMIX-Universal Server*.

## Printed Manuals

The Universal Server documentation set describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see *Getting Started with INFORMIX-Universal Server*.

To order printed manuals, call 1-800-331-1763 or send email to [moreinfo@informix.com](mailto:moreinfo@informix.com).

Please provide the following information:

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

## Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. To read the error messages, Informix provides the **Find Error** option in the Universal Server Program Group.

## Documentation Notes and Release Notes

In addition to printed documentation, the following on-line files, located in the %INFORMIXDIR%\release\en\_us\0333 directory, supplement the information in this manual.

Program Group Item	Purpose
<b>Documentation Notes</b>	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication.
<b>Release Notes</b>	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.

Please examine these files because they contain vital information about application and performance issues.

## 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, on INFORMIX-Universal Server. In addition, many features of Universal Server comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

---

## **Informix Welcomes Your Comments**

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

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

Write to us at the following address:

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

If you prefer to send email, our address is:

`doc@informix.com`

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

650-926-6571

We appreciate your feedback.

# Universal Server for Windows NT

Windows NT Threads and Universal Server Threads . . . . .	1-3
Virtual Processors Implemented as Windows NT Threads . . . . .	1-3
Virtual-Processor Classes for Windows NT . . . . .	1-5
Informix-Admin Group . . . . .	1-6
Allocating Disk Space . . . . .	1-7
Universal Server for Windows NT as a Windows NT Service . . . . .	1-8
Starting and Stopping Universal Server for Windows NT . . . . .	1-8
If Universal Server Fails to Start. . . . .	1-8
The Windows NT Registry . . . . .	1-9
How Universal Server Uses the Windows NT Registry . . . . .	1-10
The Windows NT Event Logs . . . . .	1-11
The Message Server . . . . .	1-12
Auditing . . . . .	1-12
Event Logs for Auditing. . . . .	1-13
The onaudit Utility . . . . .	1-14
Showing the Auditing Configuration . . . . .	1-15
Changing the Auditing Configuration . . . . .	1-15
The onshowaudit Utility . . . . .	1-17
Creating a Table for Audit Analysis with SQL . . . . .	1-18
The Location of Universal Server Files . . . . .	1-19



**T**his chapter describes some of the most significant ways in which the implementation of INFORMIX-Universal Server for Windows NT differs from the implementation of INFORMIX-Universal Server for UNIX.

For a detailed summary of the differences, see Appendix A, “Summary of Universal Server for Windows NT.”

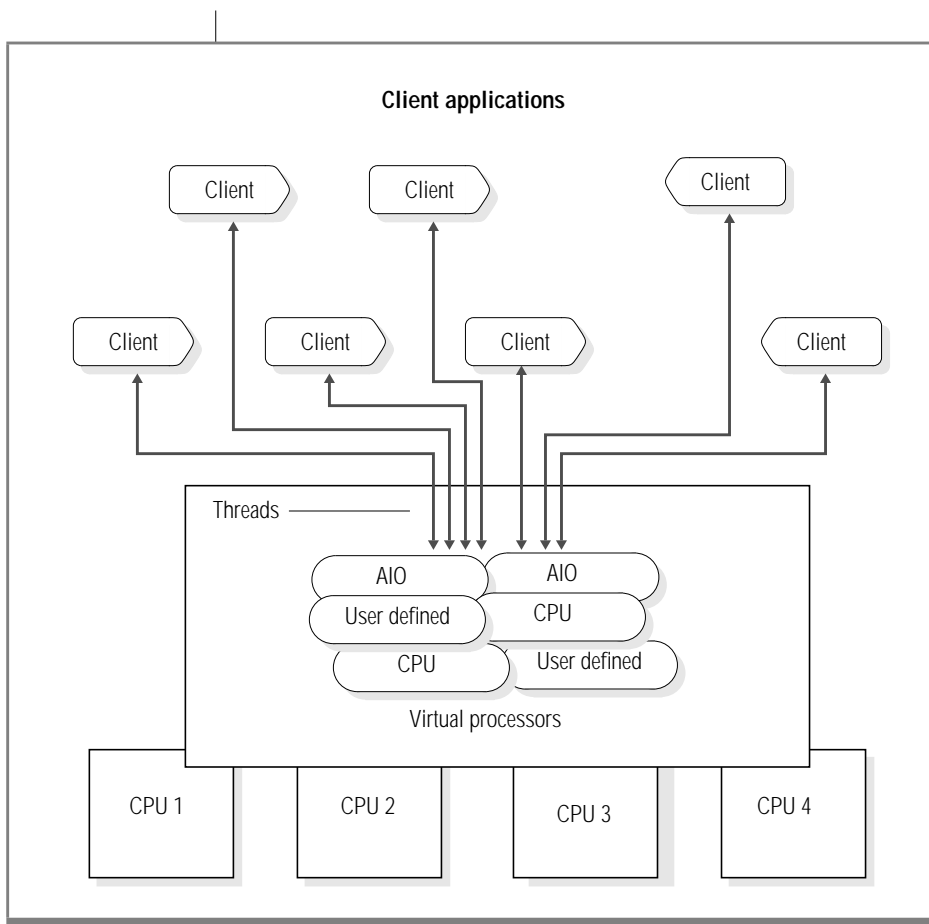
---

## Windows NT Threads and Universal Server Threads

The Windows NT operating system and Universal Server are both multi-threaded operating environments that provide concurrent service to multiple users. Universal Server for Windows NT runs within the multithreaded Windows NT environment, but it implements threads in a different manner than Universal Server for UNIX.

### Virtual Processors Implemented as Windows NT Threads

In Universal Server, a *virtual processor* is a component that is logically similar to a processor in the computer. It is the component on which processing for client applications, as well as internal processing, is scheduled and accomplished. Physically, virtual processors are implemented either as processes or as threads. In Universal Server for UNIX, virtual processors are implemented as UNIX processes as shown in Figure 1-1 on page 1-4.



**Figure 1-1**  
Virtual Processors  
Implemented as  
Windows NT  
Threads

However, in Universal Server for Windows NT, virtual processors are implemented as Windows NT *threads*. For an overall description of the Universal Server architecture and its elements, see the *INFORMIX-Universal Server Administrator's Guide*.



## Virtual-Processor Classes for Windows NT

Virtual processors are divided into *classes* that are based on the type of processing that they perform. Each class of virtual processor is dedicated to processing certain types of threads. This section shows the classes of virtual processors and the type of processing that each class performs.

Virtual-Processor Class	Category	Purpose
CPU	Database server	Runs all user threads and some system threads. Runs threads for kernel asynchronous I/O. (Can run a single poll thread, depending on configuration.)
PIO	Disk I/O	Writes to the physical-log file. (Internal class. Implemented but not used.)
LIO	Disk I/O	Writes to the logical-log files. (Internal class. Implemented but not used.)
AIO	Disk I/O	Performs disk I/O when kernel asynchronous I/O is not used. (Implemented but not used.)
KAIO	Kernel asynchronous I/O	Performs all database and logging I/O actions. (Internal class. Implemented but not used.)
SOC	Network	Performs network communication using sockets.
ADM	Administrative	Performs administrative functions.
ADT	Auditing	Performs auditing functions.
MSC	Miscellaneous	Services requests for system calls that require a very large stack. Performs miscellaneous system I/O
<i>classname</i>	User defined	Runs user-defined routines in a thread-safe manner so that if the routine fails, the database server is unaffected. Specified using the VPCLASS configuration parameter.



**Important:** Universal Server for Windows NT uses *kernel asynchronous I/O* for all database and logging I/O.

The following table lists Universal Server for Windows NT configuration parameters that affect the configuration of virtual processors. The right-hand column of the table describes the values, if any, to which you can set the parameter.

ONCONFIG Parameter	Value
AFF_NPROCS	Not used. Processor affinity is not supported.
AFF_SPROC	Not used. Processor affinity is not supported.
MULTIPROCESSOR	Set to 1 if your Windows NT computer has multiple CPUs. Set to 0 if your Windows NT computer is a single-CPU computer.
NOAGE	Not used.
NUMCPUVPS	Set to the number of CPUs plus one.
SINGLE_CPU_VP	Set to 1 if your Windows NT computer has only one CPU. Set to 0 if your Windows NT computer has multiple CPUs.

For more information about virtual-processor classes and about the configuration parameters that affect them, refer to the *INFORMIX-Universal Server Administrator's Guide*.

## Informix-Admin Group

Universal Server for Windows NT does not use the user names **informix** and **root** for Universal Server administration because Windows NT does not support multiple login sessions for the same user. Universal Server for Windows NT creates a user group called the *Informix-Admin Group*. You must assign users who administer Universal Server for Windows NT to the Informix-Admin Group. To add users to the Informix-Admin Group, choose **Administrative Tools→User Manager** from your Windows NT workstation, select the Informix-Admin Group, and add the Universal Server users.

---

## Allocating Disk Space

Universal Server for Windows NT uses only NT file system (NTFS) files for disk space. It does not use raw disk space.

Before you can create a dbspace, blobspace, or sbspace (including a mirror dbspace, sbspace, or blobspace) you must allocate the initial chunk as a null (zero bytes) file. To allocate NTFS file space for dbspaces and blobspaces, perform the following steps:

1. Log in as a member of the Informix-Admin group.
2. Open an MS-DOS command shell.
3. Change to the directory where you want to allocate the space, as in the following example:  

```
c:> cd \usr\data
```
4. Create the chunk by creating a null file with the following command:  

```
c:> copy nul my_chunk
```
5. To verify that the file was created, use the **dir** command.

Once you have allocated the file space, you can create the dbspace, sbspace, or blobspace as you normally would with **onspaces**. For the procedure to create a dbspace, see the *INFORMIX-Universal Server Administrator's Guide*.

You must also follow the preceding steps before you add a chunk to a dbspace, sbspace, or blobspace.

---

## Universal Server for Windows NT as a Windows NT Service

Universal Server is installed as a Windows NT *service*, which is roughly equivalent to a daemon process for the UNIX operating system. As a service, Universal Server can run in the background.

### Starting and Stopping Universal Server for Windows NT

You must use the Services application in the Windows NT Control Panel (in the Main program group) to start and stop Universal Server. The Services application starts Universal Server as a service that runs under the **informix** user account. The Services application also allows you to specify that Universal Server should be started automatically when Windows NT is started. Starting Universal Server automatically ensures that Universal Server is running when client applications attempt to access it.



***Tip:** Specify command-line options such as **-f** and **-iy** in the Services applet text field Startup Parameters.*

For detailed descriptions of the procedures to start and stop Universal Server, see the *INFORMIX-Universal Server Installation Guide for Windows NT, Version 9.12*.

### *If Universal Server Fails to Start*

If Universal Server fails to start, the following file might contain an explanation of why it failed:

```
(Winnt)\SYSTEM32\suserv.log
```

In this example, (Winnt) is the name of the Windows NT root directory.

---

## The Windows NT Registry

Windows NT provides a database called the *registry* that stores information about users, applications, and the configuration of the Windows NT operating system in a central location. The Windows NT registry is roughly comparable to the */etc* directory in the UNIX operating system and to the **win.ini** and **system.ini** files of Windows 3.1.

The Windows NT registry consists of the following four *subtrees*:

- HKEY\_LOCAL\_MACHINE  
This subtree contains configuration information about the local computer.
- HKEY\_USERS  
This subtree contains configuration information for the current user.
- HKEY\_CURRENT\_USER  
This subtree contains profile information about the current user.
- HKEY\_CLASSES\_ROOT  
This subtree contains software configuration information for object linking and embedding and also file-class association information.

Windows NT provides a registry editor program, **regedt32.exe**, that allows administrators to view and manipulate the information stored in the registry. For more information on how to access and manipulate the registry, see your Windows NT documentation. The following section describes how Universal Server uses the Windows NT registry.

## How Universal Server Uses the Windows NT Registry

Universal Server stores configuration information in both the Windows NT registry and in the configuration files that it traditionally has used. However, Universal Server looks for configuration information first in the registry and then, if it cannot find the information, in the appropriate configuration file. The registry provides the following advantages:

- You can access it remotely.
- You can limit access to specific users or groups.

Universal Server creates and uses the following keys in the Windows NT registry:

- HKEY\_LOCAL\_MACHINE\SOFTWARE\INFORMIX  
This key creates the registry hive that stores information about Universal Server.
- HKEY\_LOCAL\_MACHINE\SOFTWARE\INFORMIX\SQLHOSTS  
This key stores **sqlhosts** information. For more information about **sqlhosts**, see “The sqlhosts Registry” on page 2-8.
- HKEY\_LOCAL\_MACHINE\SOFTWARE\INFORMIX\OnLine  
This key stores Universal Server configuration data.
- HKEY\_LOCAL\_MACHINE\SOFTWARE\INFORMIX\OnLine\Current Version  
This key stores miscellaneous information about Universal Server such as the install date, company name, software type, and so on.
- HKEY\_LOCAL\_MACHINE\SOFTWARE\INFORMIX\OnLine\Current Version\Environment  
This key stores the values of all the environment variables that are passed to Universal Server when it starts up. In particular, this key stores the values of the %INFORMIXDIR% and %INFORMIX-SERVER% environment variables.
- HKEY\_LOCAL\_MACHINE\SOFTWARE\Informix\OnLine\Current Version\Users  
This key stores the mapping of user names to user IDs. *Only Universal Server should access this key. Do not attempt to add any user names or user IDs manually.*

- HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\OnLine

The Windows NT Service Control Manager (SCM) creates this key for Universal Server, which runs as a service under user **informix**.

- HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\MsgServ

The Windows NT Service Control Manager (SCM) creates this key for the Universal Server Message Server. The Universal Server Message Server passes system, audit, and event alarm messages from Universal Server to the Windows NT event logs. For more information about the Message Server service, see “The Message Server” on page 1-12.

---

## The Windows NT Event Logs

The Windows NT operating system provides an event-logging facility as a common repository for storing logs and other useful information. The event-logging facility also provides a user interface to filter, view, and back up the information that is stored there.

Windows NT provides secure event logs, so Universal Server does not need to provide any additional security. Security logs and system logs are for use only by Windows NT services that are running under the **LocalSystem** user account and the Windows NT security subsystem.

Any workstation in the Windows NT network can view the event logs as long as the user has sufficient access rights. Security logs are accessible only to users who belong to the Windows NT Administrator group, including Domain Administrators.

Any messages that Universal Server writes to the Universal Server log file, it also writes to the Windows NT event logs. Universal Server also writes auditing records to the event logs. For more information on auditing records and the event logs, see “Event Logs for Auditing” on page 1-13.

## The Message Server

Universal Server for Windows NT runs as a service under the **informix** user account. Because the Windows NT security logs and system logs are for use only by services running under the **LocalSystem** user account, Universal Server includes a Message Server service that runs under that account. The Message Server communicates with Universal Server through the *named pipes* interprocess communication (IPC) mechanism to receive information and write it to the event logs.

Universal Server starts the Message Server when it first needs to write a message to the event logs.

The Message Server terminates automatically when Universal Server terminates.

---

## Auditing

The Universal Server secure-auditing facility is described in the *INFORMIX-Universal Server Trusted Facility Manual*. However, the implementation of the auditing facility for Universal Server for Windows NT differs from what is described in that manual in the following ways:

- Universal Server for Windows NT uses the Windows NT event logs instead of audit-trail files to record audit events.
- Universal Server uses a Message Server to receive messages from Universal Server and write them to the event logs.
- Universal Server does not use the ADTPATH and ADTSIZE parameters (because it does not use audit-trail files).
- The **onaudit** utility does not support the following options:
  - **-n** (start a new audit file)
  - **-p auditdir** (specify the directory where the audit files are to be created)
  - **-s maxsize** (specify the maximum size of an audit file)

The **-c** option, which displays the current configuration parameters, displays only the values of the ADTMODE and ADTERR parameters.



- The **onshowaudit** utility does not support the **-I** (Informix-maintained audit files) and **-O** (operating-system-maintained audit files) options because the Windows NT operating system handles all auditing functions.
- The **onshowaudit** utility supports the following additional options:
  - **-ts** to extract only *success audit* records
  - **-tf** to extract only *failure audit* records
  - **-d** to assume default values for the user (current user) and the database server (INFORMIXSERVER)

The **-ts** and **-tf** options are mutually exclusive, and the **-u username** and **-s servername** options are mutually exclusive with respect to the **-d** (default) option. For descriptions of these options, see “The onshowaudit Utility” on page 1-17.

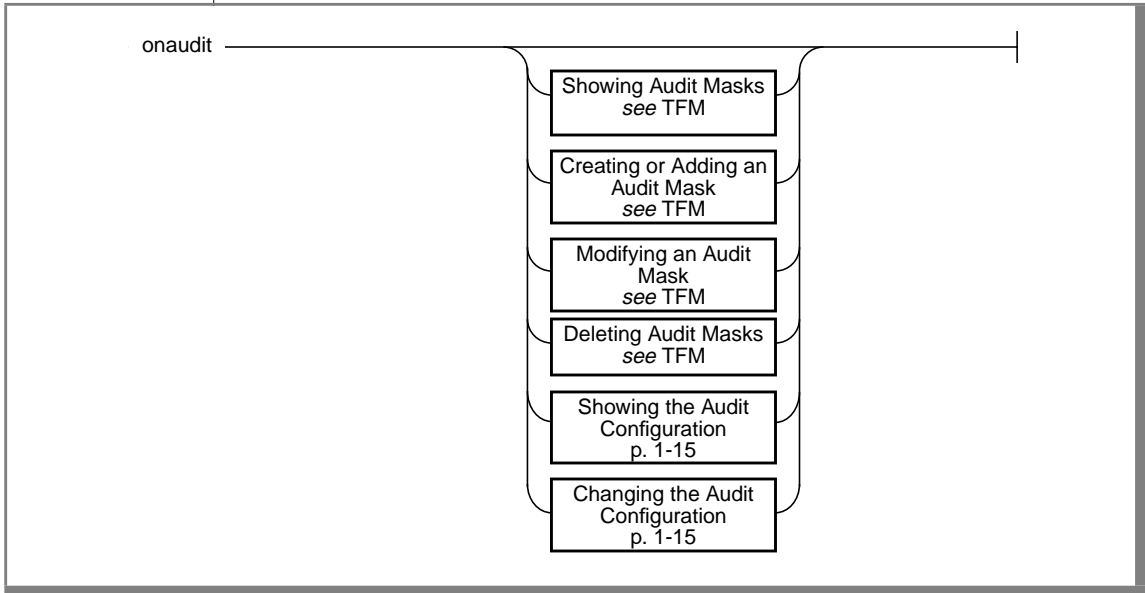
## Event Logs for Auditing

Universal Server uses the event-logging facility, instead of audit-trail files, to store audit event records. The format of audit-trail records in the event logs is the same as for records in standard Universal Server audit-trail files, as the *INFORMIX-Universal Server Trusted Facility Manual* describes.

The **onshowaudit** utility enables you to filter and review the Universal Server audit records stored in the event logs. For a description of **onshowaudit**, see “The onshowaudit Utility” on page 1-17.

## The onaudit Utility

This section and the following syntax diagram illustrate the **onaudit** tasks available in the Windows NT version of the utility.



The syntax diagrams in the following sections illustrate the **onaudit** options for showing and changing the auditing configuration under Windows NT. The other four tasks that **onaudit** performs affect audit masks, and they are described in the *INFORMIX-Universal Server Trusted Facility Manual*.

## Showing the Auditing Configuration

Under Windows NT, the **-c** option displays only the values of the ADTMODE and the ADTERR parameters.

Showing the Auditing Configuration

→ **-c** →

**-c** shows the current auditing configuration.

The **-c** option directs the **onaudit** utility to display the current state of auditing. Figure 1-2 shows sample audit-configuration output.

```
onaudit -c

Onaudit -- Audit Subsystem Control Utility
Copyright (c) Informix Software, Inc., 1995

Current audit system configuration:
    ADTMODE      = 1
    ADTERR       = 0
```

**Figure 1-2**  
Sample  
Audit-Configuration  
Output

## Changing the Auditing Configuration

Under Windows NT, the **onaudit** options available for changing the audit configuration are the **-l** and **-e** options.

Changing the Audit Configuration

→ **-l audit mode** **-e error mode** →

<b>-l audit mode</b>	<p>specifies the auditing mode. This option pertains to the value set for the ADTMODE parameter in the ADTCFG file. It can have one of the following values:</p> <ul style="list-style-type: none"><li>0 turns auditing off. Universal Server stops auditing for all existing sessions, and new sessions are not audited.</li><li>1 turns on Universal Server-managed auditing; it does not automatically audit DBSSO or administrator actions.</li><li>3 turns on Universal Server-managed auditing; it automatically audits DBSSO actions.</li><li>5 turns on Universal Server-managed auditing; it automatically audits administrator actions.</li><li>7 turns on Universal Server-managed auditing; it automatically audits DBSSO and administrator actions.</li></ul>
<b>-e error mode</b>	<p>specifies the error-handling method for auditing when Universal Server cannot write a record to the audit file. This option specifies the value of the ADTERR parameter in the ADTCFG file and is in effect only when auditing is on. The <i>error mode</i> can have one of the following values:</p> <ul style="list-style-type: none"><li>0 a <i>halt mode</i>; it indicates that Universal Server should suspend processing a thread when it cannot write a record to the current audit file and should continue the write attempt until it succeeds.</li><li>1 also known as <i>continue mode</i>; it indicates that Universal Server should continue processing the thread and note the error in the message log. Errors for subsequent attempts to write to the audit file are also sent to the message log.</li><li>2 a <i>halt mode</i>; it indicates that Universal Server is to terminate the session.</li><li>3 a <i>halt mode</i>; it indicates that Universal Server is to shut down.</li></ul>

## The onshowaudit Utility

The **onshowaudit** utility for Windows NT includes the **-ts** and **-tf** options to show only *success* or only *failure* audit records and the **-d** option to assume default values for the *username* and *servername*. The Windows NT version of **onshowaudit** does not support the **-I** or the **-O** options because **onshowaudit** uses the Windows NT event logs to store audit-trail records.

onshowaudit

-ts

-tf

-u username

-s servername

-I

-d

- ts** directs **onshowaudit** to show only *success* audit records.
- tf** directs **onshowaudit** to show only *failure* audit records.
- d** assumes the default values for the user (*current user*) and the database server (INFORMIXSERVER).
- u username** specifies the login name of a user about whom to extract audit information.
- s servername** names the specific database server about which to extract audit information.
- I** directs **onshowaudit** to take the extracted information and reformat it for use by **dbload**. For information on the file format, see the *INFORMIX-Universal Server Trusted Facility Manual* and also “Creating a Table for Audit Analysis with SQL” on page 1-18 in the following section. For information on the **dbload** utility, see the *Informix Guide to SQL: Reference*.

### ***Creating a Table for Audit Analysis with SQL***

The *INFORMIX-Universal Server Trusted Facility Manual* describes how to perform audit analysis with SQL. However, when you use Universal Server, you must use the specifications of the CREATE TABLE statement shown in Figure 1-3 to create the table that receives the audit data from **dbload**.

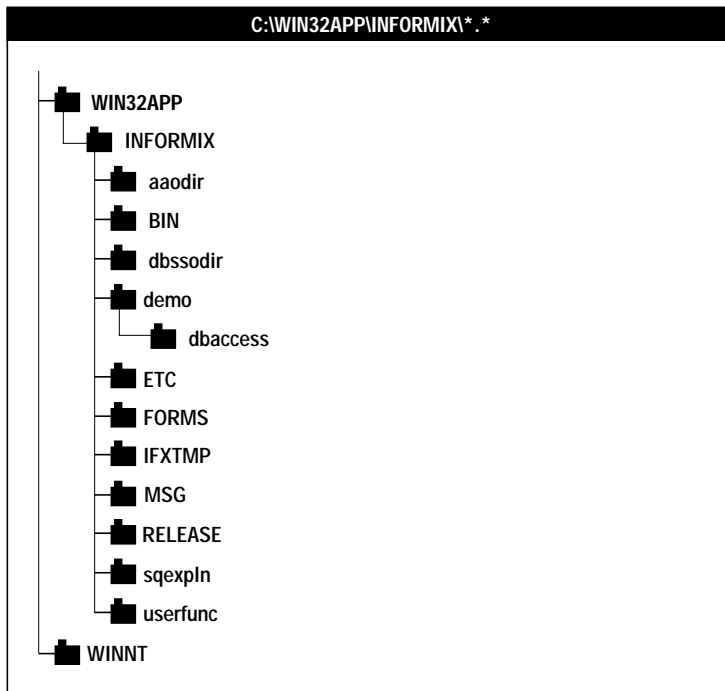
```
CREATE TABLE april95_tab(  
    adttag CHAR(4),  
    date_time DATETIME year TO fraction(3),  
    hostname CHAR(18),  
    pid INT,  
    server CHAR(32),  
    username CHAR(18),  
    errno INT,  
    code CHAR(4),  
    dbname CHAR(18),  
    tabid INT,  
    objname CHAR(18),  
    extra_1 INT,  
    partno INT,  
    row_num INT,  
    login CHAR(8),  
    flags INT,  
    extra_2 VARCHAR(160,1));
```

**Figure 1-3**  
*Sample CREATE  
TABLE Statement  
for Audit Table*

## The Location of Universal Server Files

The installation program creates the folders shown in Figure 1-4 when it installs Universal Server.

**Figure 1-4**  
*Universal Server  
Folders*



The following table describes the contents of the Universal Server folders shown in Figure 1-4 on page 1-19.

Folder	Content
aaodir	The <b>adtcfg</b> audit-configuration file and the <b>adtcfg.std</b> file (For information about these files, see the <i>INFORMIX-Universal Server Trusted Facility Manual</i> .)
BIN	Executable files and programs
dbssodir	Audit mask and security configuration files (For information about these files, see the <i>INFORMIX-Universal Server Trusted Facility Manual</i> .)
demo	DB-Access folder
demo/dbaccess	SQL scripts and unloaded <b>stores7</b> database tables
ETC	Configuration and system files, including the ONCONFIG file
FORMS	DB-Access screen forms
INFXTMP	Temporary system files
MSG	Informix message files
RELEASE	Release Notes for this release
sqexpln	Output files for SET EXPLAIN statement
userfunc	Location of user-defined routines



---

# Client/Server Communications

Universal Server and Windows NT Network Domains . . . . .	2-3
Connecting to Universal Server for Windows NT . . . . .	2-4
Connecting Different Types of Clients . . . . .	2-5
Creating a TCP/IP Connection to Universal Server . . . . .	2-6
Resolving Internet Domain Addresses . . . . .	2-6
Authenticating Users . . . . .	2-6
The sqlhosts Registry . . . . .	2-8
How the sqlhosts Registry Is Created . . . . .	2-8
The Location of the sqlhosts Registry . . . . .	2-11
Changing the sqlhosts Registry . . . . .	2-12



**T**his chapter describes the implementation of Universal Server for Windows NT client/server communications. It briefly describes aspects of Windows NT networks that affect client applications connecting to Universal Server. It also describes the basic requirements to create a TCP/IP connection to Universal Server and describes how **sqlhosts** information is stored and maintained under Windows NT.

---

## Universal Server and Windows NT Network Domains

Windows NT network technology enables you to create network *domains*. A domain is a group of connected Windows NT computers that share user account information and a security policy. A *domain controller* manages the user account information for all domain members.

The domain controller facilitates network administration. By managing one account list for all domain members, the domain controller relieves the network administrator of the requirement to synchronize the account lists on each of the domain computers. In other words, when the network administrator creates or changes a user account, he or she needs to update only the account list on the domain controller rather than the account lists on each of the computers in the domain.

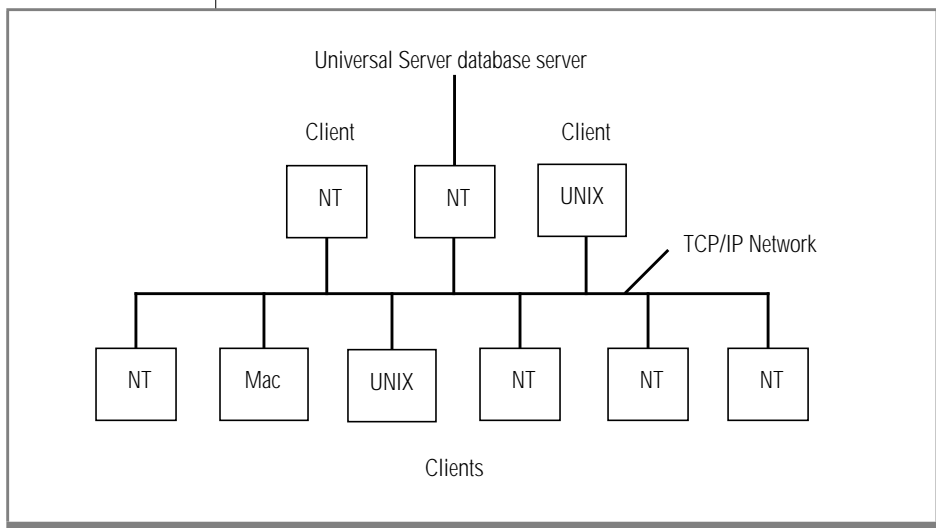
To log in to a Windows NT server, a user on another Windows NT computer must belong to either the same domain or a *trusted domain*. A trusted domain is one that has established a *trust relationship* with another domain. In a trust relationship, user accounts are located only in the trusted domain, but users can log on to the trusting domain.



**Important:** The Universal Server trusted client mechanism has nothing to do with the trust relationship that you can establish between Windows NT domains. Therefore, even if a Universal Server client connects from a trusted Windows NT domain, Universal Server requires the user to have an account in the domain on which Universal Server is running. For more information on how Universal Server authenticates clients, see “Authenticating Users” on page 2-6.

## Connecting to Universal Server for Windows NT

As Figure 2-1 shows, client applications that run on Windows NT, UNIX, and Apple Macintosh computers can all connect to Universal Server for Windows NT.



**Figure 2-1**  
Universal Server for  
Windows NT  
Serving Client  
Applications on  
Windows NT, UNIX,  
and Macintosh  
Computers

A client application can connect with Universal Server in the following two ways:

- Through a TCP/IP network connection from another computer
- Through a *local loopback connection* on the computer where Universal Server is running

A local loopback connection mimics a TCP/IP network connection by using the same technology. Communication does not go out to the network, however. It simply flows between the client and Universal Server on the same computer. For a comprehensive explanation of how client applications connect to and communicate with Universal Server, see the *INFORMIX-Universal Server Administrator's Guide*.

Universal Server for Windows NT uses the Windows Sockets interface to support the TCP/IP communication protocol for client-to-server and server-to-server connections.

Universal Server for Windows NT also supports the OSI TP4 transport for server-to-server connections only. For more information on implementing the OSI TP4 transport, see Appendix B, "Using OSI TP4."

## Connecting Different Types of Clients

The general requirements for implementing a TCP/IP connection to Universal Server are described in the following section, "Creating a TCP/IP Connection to Universal Server" on page 2-5. However, the procedures to implement a TCP/IP connection for each type of client computer vary. Consult the Informix documentation for the particular type of client application that you want to connect to Universal Server. For example, to connect an ESQL/C application on an Apple Macintosh computer, consult the *INFORMIX-ESQL/C Programmer's Supplement*.

## Creating a TCP/IP Connection to Universal Server

To create a TCP/IP connection to Universal Server, perform the following steps:

1. Configure the network interface (or adapter) card.
2. Install and configure the Windows NT TCP/IP network software package.
3. Create an entry in the Universal Server **sqlhosts** registry.

For information on installing a network interface card for TCP/IP, consult your computer hardware documentation. To configure the network card and to install and configure the TCP/IP network software, use the Network application in the Windows NT Control Panel. For specific directions on configuring the adapter and installing the TCP/IP network software, see your Windows NT system documentation.

For information on the **sqlhosts** registry, see “The sqlhosts Registry” on page 2-8.

## Resolving Internet Domain Addresses

You can configure Windows NT to use either of the following mechanisms for resolving Internet Domain Addresses (*mymachine.informix.com*) to Internet Protocol addresses (149.8.73.14):

- Windows Internet Name Service
- Domain Name Server (UNIX Yellow Pages)

Universal Server operates equally well in configurations that use either of these name-resolution services. Both services create an entry in the **hosts** file.

## Authenticating Users

All client users, whether connecting from Windows NT, UNIX, or other computers, must have a user account in the Windows NT domain on which Universal Server runs. Universal Server for Windows NT needs the client user's name and password to impersonate the user when it runs tasks on the user's behalf.

### *The hosts.equiv file*

You can create a **hosts.equiv** file on Windows NT to specify trusted Universal Server clients who can connect from remote computers. The **hosts.equiv** file specifies trusted users who are allowed to access Universal Server without supplying a password. For more information about the **hosts.equiv** file, see the *INFORMIX-Universal Server Administrator's Guide*. If you create a **hosts.equiv** file, place it in the `\(Winnt)\system32\drivers\etc` directory where **(Winnt)** is the root directory that you specified for Windows NT.

If the client application supplies an invalid account name and password, Universal Server rejects the connection even if the **hosts.equiv** file contains an entry for the client computer. Use the **hosts.equiv** file only for client applications that do not supply a user account or password.

### *The CONNECT Statement*

Your client application can supply a user account and password through the USER clause of the CONNECT statement. The USER clause of the CONNECT statement takes higher precedence than other forms of authentication that the client can use.

### *UNIX Clients: The ~/.netrc File*

If you access Universal Server for Windows NT from a UNIX client, you can supply a user name and password through an entry in the **.netrc** file, which you can create in your home directory. If you use a different user ID and password on the Windows NT computer, you can create an entry for it in the **.netrc** file. For more information about the **.netrc** file, see the *INFORMIX-Universal Server Administrator's Guide* and your UNIX documentation.

---

## The sqlhosts Registry

In addition to configuring an adapter card and the TCP/IP network software, you must create an entry in the Universal Server **sqlhosts registry** to enable client/server communications. The **sqlhosts** registry is the same mechanism as the Universal Server **sqlhosts file** on a UNIX computer. The information is simply stored in the Windows NT registry instead of in a file.

For the following reasons, the Windows NT registry makes an ideal location for storing **sqlhosts** information:

- The registry can be administered remotely.
- The registry is the standard Windows NT location for storing configuration information.

## How the sqlhosts Registry Is Created

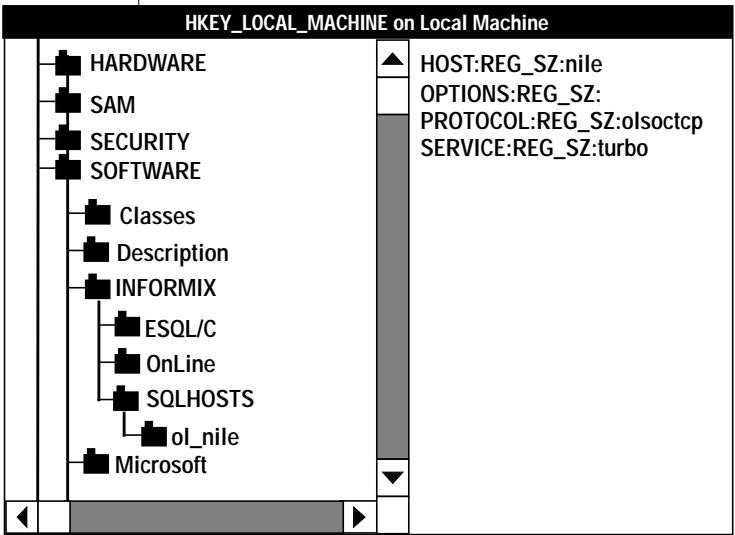
When you install Universal Server, the **setup** program creates the following key in the Windows NT registry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\INFORMIX\SQLHOSTS
```

This branch of the HKEY\_LOCAL\_MACHINE subtree stores the **sqlhosts** subtree. The name of Universal Server is a key on the \SOFTWARE\INFORMIX\SQLHOSTS branch. If you click the Universal Server name, the registry displays the values of the HOST, OPTIONS, PROTOCOL, and SERVICE keys that make up the **sqlhosts** entry for that particular database server. These values correspond to the **hostname** (HOST), **options** (OPTIONS), **nettype** (PROTOCOL), and **servicename** (SERVICE) fields of an entry in the **sqlhosts** file on a UNIX system.



The values that you can enter for these fields are described in the *INFORMIX-Universal Server Administrator's Guide*. Figure 2-2 illustrates the location and content of the **sqlhosts** registry for the sample Universal Server **ol\_nile**.

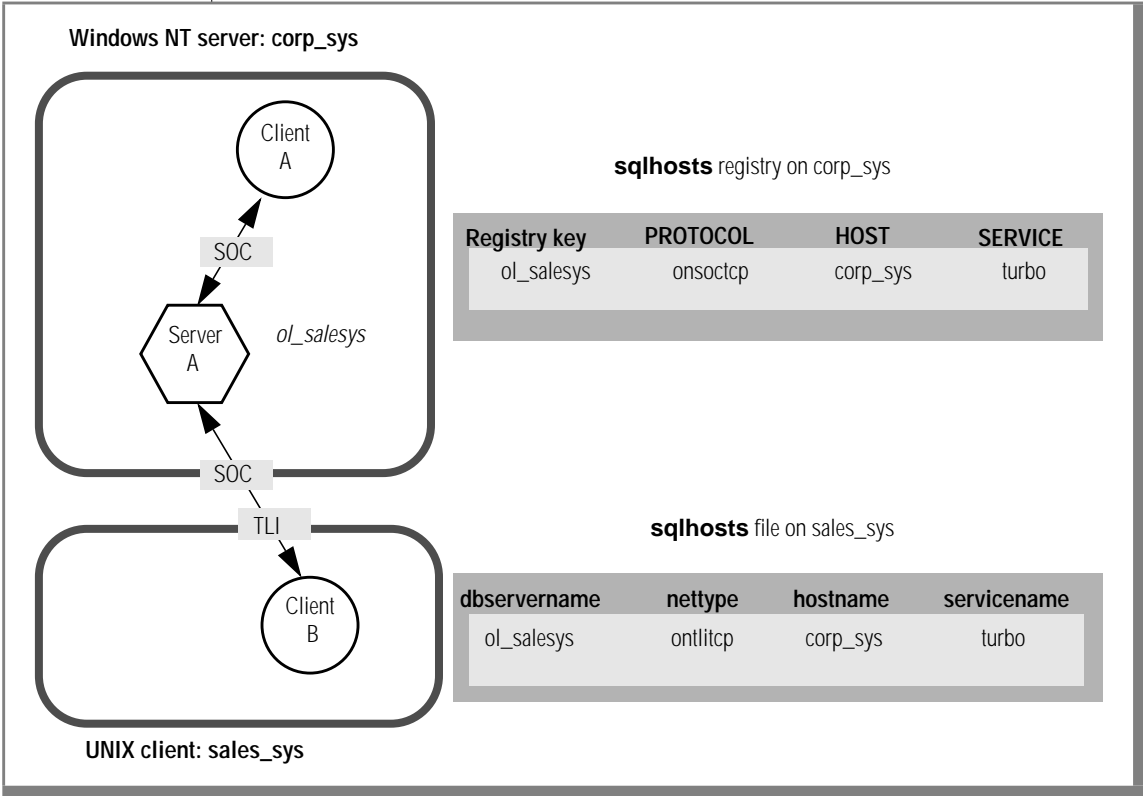


**Figure 2-2**  
*sqlhosts Information in  
the Windows NT  
Registry*

Figure 2-3 on page 2-10 illustrates the **sqlhosts** values for a local loopback client and a network client. The entries are the same on both systems because both clients use the same protocol (or **nettype**), and the remaining values refer to the same Universal Server server.

Figure 2-3

Example of sqlhosts Entries for Local Loopback and Network Client TCP/IP Connections



When the client application runs on the same computer as Universal Server, they share a single **sqlhosts** registry.

The **PROTOCOL** field of the **sqlhosts** registry on the computer where Universal Server resides must specify a value of either **onsoctcp** or **olsoctcp**.

A UNIX *client* can use either the sockets interface (SOC) or the transport layer interface (TLI) to connect to Universal Server for Windows NT, depending on which TCP/IP interface the UNIX client supports.

## The Location of the sqlhosts Registry

When you install Universal Server, you have the option of specifying the name of the computer where you want to store the **sqlhosts** registry. You can specify the location as one of the following two options:

- The local computer where you are installing Universal Server
- Another computer in the network that serves as a known, central repository for **sqlhosts** information about multiple Universal Server database servers in the network

If you use a central **sqlhosts** registry, you must set the **INFORMIXSQLHOSTS** environment variable on your local computer to the name of the Windows NT computer that stores the registry. Using a central **sqlhosts** registry relieves you of the necessity to maintain the same **sqlhosts** information on multiple computers.

You must comply with Windows NT network-access conventions and file permissions to ensure that the local computer has access to the central **sqlhosts** registry. For information about network-access conventions and file permissions, see your Windows NT documentation.

Universal Server looks for the **sqlhosts** registry on the **INFORMIXSQLHOSTS** computer first. If Universal Server does not find an **sqlhosts** registry on the **INFORMIXSQLHOSTS** computer, or if **INFORMIXSQLHOSTS** is not set, Universal Server looks for an **sqlhosts** registry on the local computer.

## Changing the sqlhosts Registry

You must use the Windows NT program **regedt32** to change an entry in the **sqlhosts** registry.

### To change an entry

1. Run **regedt32**.
2. In the Registry Editor window, select the window for the HKEY\_LOCAL\_MACHINE subtree.
3. Click the folder icons to select the \SOFTWARE\INFORMIX\SQLHOSTS branch.
4. Click the folder icon for the name of the database server.
5. On the right-hand side of the split window, double-click the **sqlhosts** value that you want to edit (HOST, OPTIONS, and so on).
6. In the String Editor box, type the new value for the key that you selected and click **OK**.
7. Repeat steps 5 and 6 for each value that you want to change.
8. Exit the Registry Editor.

# Tables and Fragmentation

Placement of Tables on Disk. . . . .	3-3
Isolating High-Use Tables . . . . .	3-5
Using Multiple Disks for a Dbspace. . . . .	3-6
Improving Smart-Large-Object Metadata I/O . . . . .	3-6
Spreading Temporary Tables and Sort Files Across Multiple Disks. . . . .	3-7
Backup and Restore Considerations. . . . .	3-8
Improving Performance for Nonfragmented Tables and Table Fragments . . . . .	3-8
Estimating the Size of a Table and Index . . . . .	3-8
Estimating Data Pages . . . . .	3-9
Estimating Index Pages. . . . .	3-13
Estimating Pages Occupied by Smart Large Objects . . . . .	3-24
Estimating Pages Occupied by Simple Large Objects . . . . .	3-27
Locating Simple-Large-Object Data in the Tblspace or a Separate BlobSpace . . . . .	3-28
Managing Indexes. . . . .	3-29
Costs Associated with Indexes . . . . .	3-30
Maintaining Indexes. . . . .	3-32
Dropping Indexes . . . . .	3-33
Improving Performance for Index Builds . . . . .	3-34
Managing Extents . . . . .	3-36
Choosing Table Extent Sizes . . . . .	3-37
Choosing Index Extent Sizes . . . . .	3-39
Upper Limit on Extents . . . . .	3-40
Checking for Extent Interleaving . . . . .	3-41
Eliminating Interleaved Extents. . . . .	3-41
Reclaiming Unused Space Within an Extent . . . . .	3-44
Managing Smart Large Objects . . . . .	3-45

Denormalizing the Data Model to Improve Performance . . . . .	3-46
Using Shorter Rows for Faster Queries . . . . .	3-46
Expelling Long Strings . . . . .	3-47
Using VARCHAR Strings . . . . .	3-47
Using TEXT Simple Large Objects . . . . .	3-47
Moving Strings to a Companion Table . . . . .	3-47
Building a Symbol Table . . . . .	3-48
Splitting Wide Tables . . . . .	3-48
Division by Bulk . . . . .	3-49
Division by Frequency of Use . . . . .	3-49
Division by Frequency of Update . . . . .	3-49
Costs of Companion Tables . . . . .	3-49
Redundant Data . . . . .	3-50
Adding Redundant Data . . . . .	3-50
Table-Fragmentation Guidelines . . . . .	3-51
Considering Fragmentation Strategy . . . . .	3-54
Examining Your Data and Queries . . . . .	3-55
Fragmentation Goals . . . . .	3-56
Improving Performance for Individual Queries . . . . .	3-56
Reducing Contention Between Queries . . . . .	3-57
Increasing Availability of Data . . . . .	3-58
Increasing Granularity for Backup and Restore . . . . .	3-59
Distribution Scheme . . . . .	3-60
Choosing Between Round-Robin and Expression-Based	
Distribution Schemes . . . . .	3-61
Designing an Expression-Based Distribution Scheme . . . . .	3-62
Monitoring Fragment Use . . . . .	3-64
Fragmenting Indexes . . . . .	3-64
Improving Fragmentation . . . . .	3-65

**T**his chapter describes how to organize tables on disk to improve performance. It covers the following topics:

- Placement of tables on disk to increase throughput and reduce contention
- Layout of tables for more efficient access
- Denormalization of tables, when necessary, to reduce overhead
- Fragmentation of tables to take advantage of multiple disks and parallel database queries (PDQ)

---

## Placement of Tables on Disk

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

When you configure chunks and allocate them to dbspaces, you must relate the size of your dbspaces to the tables or fragments that each dbspace is to contain. To estimate the size of a table, follow the instructions in “Estimating the Size of a Table and Index” on page 3-8.

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

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

The DBA can fragment a table across multiple dbspaces, as described under “Table-Fragmentation Guidelines” on page 3-51, or move a table to another dbspace using the ALTER FRAGMENT statement. The ALTER FRAGMENT statement provides the simplest way to alter the table placement. However, the table is unavailable while Universal Server processes the alteration. Schedule the movement of a table or fragment at a time that affects the fewest users. The ALTER FRAGMENT statement is described in the *Informix Guide to SQL: Syntax*.

You can move tables between dbspaces in other ways. A DBA can unload the data from a table and then move that data to another dbspace with the LOAD and UNLOAD SQL statements, as described in the *Informix Guide to SQL: Syntax*. The administrator can perform the same actions with the **onload** and **onunload** utilities, as described in the *INFORMIX-Universal Server Administrator's Guide*.

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

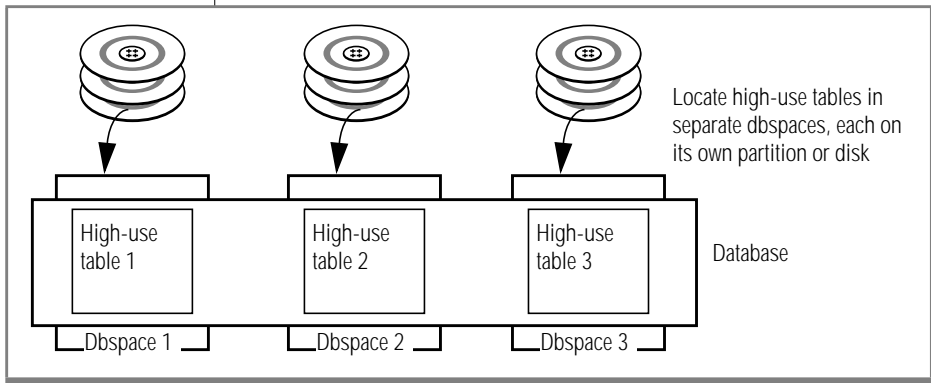
Depending on the size, fragmentation strategy, and indexes that are associated with a table, it can be faster to unload a table and reload it than to alter fragmentation. For other tables, it can be faster to alter fragmentation. You might have to experiment to determine which method is faster for a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



## Isolating High-Use Tables

You can place a table with high I/O activity on a dedicated disk device and thus reduce contention for the data that is stored in that table. When disk drives have different performance levels, you can put the tables with the highest use on the fastest drives. Placing two high-use tables on separate disk devices reduces competition for disk access when the two tables experience frequent, simultaneous I/O from multiple applications or when joins are formed between them.

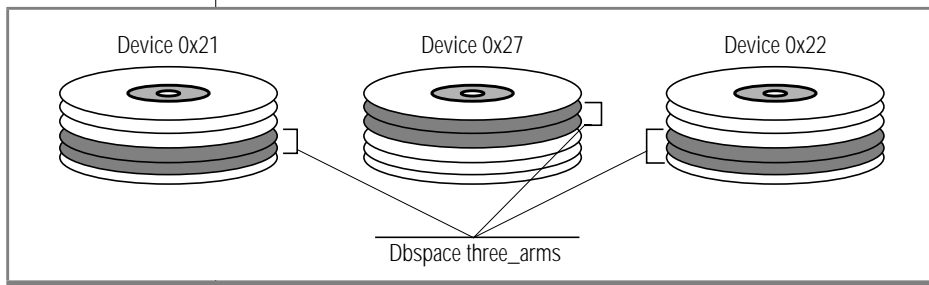
To isolate a high-use table on its own disk device, assign the device to a chunk, assign that chunk to a dbspace, and then place the table in the dbspace that you created. Figure 3-1 shows three tables with high use placed on three disks.



**Figure 3-1**  
Isolating High-Use  
Tables

## Using Multiple Disks for a Dbspace

A dbspace can be composed of multiple chunks, and each chunk can represent a different disk. This arrangement allows you to distribute data in a dbspace over multiple disks. Figure 3-2 shows a dbspace distributed over multiple disks.



**Figure 3-2**  
*A Dbspace  
Distributed over  
Three Disks*

Use multiple disks for a dbspace to help distribute I/O across dbspaces that contain several small tables. Because this type of distributed dbspace cannot be used for parallel database queries (PDQ), Informix recommends that you use the table-fragmentation techniques described in “Table-Fragmentation Guidelines” on page 3-51 to partition large, high-use tables across multiple dbspaces.

## Improving Smart-Large-Object Metadata I/O

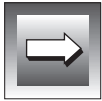
The metadata pages in an sbspace contain information about the location of the smart large objects in the sbspace. Typically, these pages are read intensive. You can distribute I/O to these pages in one of three ways:

- Mirror the chunks that contain metadata.

For more information on mirroring implications, refer to the *INFORMIX-Universal Server Administrator's Guide*.

- Position the metadata pages on the fastest portion of the disk.

Because the metadata pages are the most read-intensive part of an sbspace, you should place the metadata pages toward the middle of the disk to minimize disk seek time. To position metadata pages, use the **-Mo** option when you create the sbspace or add a chunk with the **onspace** utility.



- Spread metadata pages across disks.

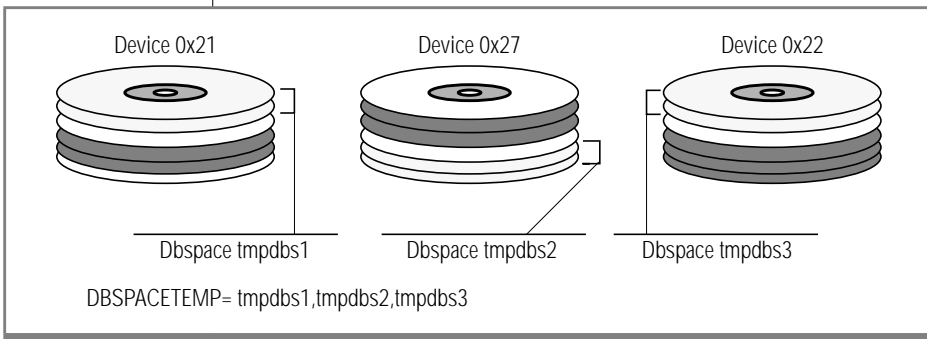
To spread metadata pages across disks, create multiple chunks in an sbspace, with each chunk residing on a separate disk. When you add a chunk to the sbspace with the **onspaces** utility, specify the **-Ms** option to allocate pages for the metadata information.

Although Universal Server attempts to keep the metadata information with its corresponding data in the same chunk, it cannot guarantee that they remain together.

**Important:** For highest data availability, mirror all sbspace chunks that contain metadata.

## Spreading Temporary Tables and Sort Files Across Multiple Disks

To define several dbspaces for temporary tables and sort files, use **onspaces -t**. When you place these dbspaces on different disks and list them in the **DBSPACETEMP** configuration parameter, you can spread the I/O associated with temporary tables and sort files across multiple disks, as Figure 3-3 illustrates. Dbspaces that contain regular tables can also be listed in **DBSPACETEMP**.



**Figure 3-3**  
*Dbspaces for  
Temporary Tables  
and Sort Files*

Users can specify their own lists of dbspaces for temporary tables and sort files with the **DBSPACETEMP** environment variable. Users can specify directories for sort files with the **PSORT\_DBTEMP** environment variable. For details, refer to the *INFORMIX-Universal Server Administrator's Guide*.

## Backup and Restore Considerations

When you decide where to place your tables or fragments, remember that if a device that contains a dbspace fails, all tables or table fragments in that dbspace are rendered inaccessible, even though tables and fragments in other dbspaces are accessible. The need to limit data unavailability in the event of a disk failure might influence which tables you group together in a particular dbspace.

Although you must perform a cold restore if a dbspace that contains critical data fails, you need to perform a warm restore only if a noncritical dbspace fails. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical media. For a description of a cold restore, see “Cold, Warm, or Mixed Restore—Choosing a Universal Server Mode” on page 5-44.

---

## Improving Performance for Nonfragmented Tables and Table Fragments

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

- The placement of the table or fragment, as described in previous sections
- The size of the table or fragment
- The indexing strategy used
- The size and placement of table extents with respect to one another

## Estimating the Size of a Table and Index

This section discusses how to calculate the approximate sizes (in disk pages) of tables and indexes.

The disk pages allocated to a table are collectively referred to as a *tblspace*. The *tblspace* includes data pages and index pages. If simple large objects that are not stored in an alternative dbspace are associated with the table, pages that hold simple-large-object data are also included in the *tblspace*.

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

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



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

## Estimating Data Pages

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

### Estimating Tables with Fixed-Length Rows

To estimate the size of a table with fixed-length rows in pages, perform the following steps. A table with fixed-length rows has no columns of data type VARCHAR, NVARCHAR, or LVARCHAR.

**To estimate the page size, row size, and number of rows**

1. Use **oncheck -pr** to obtain the size of a page.
2. Subtract 28 from this amount to account for the header that appears on each data page. The resulting amount is referred to as *pageuse*.
3. To calculate the size of a row, add the width of each column in the table definition. TEXT and BYTE columns each use 56 bytes. If you have already created your table, you can obtain the size of a row with the following SQL statement:

```
SELECT rowsize FROM systables WHERE tablename = 'table-name';
```



4. Estimate the number of rows that the table is expected to contain. This number is referred to as *rows*.

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

**Important:** Although the maximum size of a row that Universal Server accepts is approximately 32 kilobytes, performance degrades when a row exceeds the size of a page. For information on how to break up wide tables for improved performance, refer to “Denormalizing the Data Model to Improve Performance” on page 3-46.

5. If the size of the row is less than or equal to *pageuse*, use the following formula to calculate the number of data pages. The **trunc()** function notation indicates that you are to round down to the nearest integer.

$$\text{data\_pages} = \text{rows} / \text{trunc}(\text{pageuse} / (\text{rowsize} + 4))$$

The maximum number of rows per page is 255, regardless of the size of the row.

6. If the size of the row is greater than *pageuse*, Universal Server divides the row between pages. The page that contains the initial portion of a row is called the *home page*. Pages that contain subsequent portions of a row are called *remainder pages*. If a row spans more than two pages, some of the remainder pages are completely filled with data from that row. When the trailing portion of a row uses less than a page, it can be combined with the trailing portions of other rows to fill out the partial remainder page. The number of data pages is the sum of the home pages, the full remainder pages, and the partial remainder pages.

- a. Calculate the number of home pages. The number of home pages is the same as the number of rows.

$$\text{homepages} = \text{rows}$$

- b. Calculate the number of full remainder pages. First calculate the size of the row remainder with the following formula:

$$\text{remsize} = \text{rowsize} - (\text{pageuse} + 8)$$

If *remsize* is less than *pageuse* - 4, you have no full remainder pages. Otherwise, you can use *remsize* in the following formula to obtain the number of full remainder pages:

$$\text{fullrempages} = \text{rows} * \text{trunc}(\text{remsize} / (\text{pageuse} - 8))$$

- c. Calculate the number of partial remainder pages. First, calculate the size of a partial row remainder left after you accounted for the home and full remainder pages for an individual row. In the following formula, the **remainder()** function notation indicates that you are to take the remainder after division:

$$\text{partremsize} = \text{remainder}(\text{rowsize} / (\text{pageuse} - 8)) + 4$$

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

$$\text{partratio} = \text{partremsize} / \text{pageuse}$$

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

partratio	Formula to calculate the number of partial remainder pages
Less than .1	$\text{partrempages} = \text{rows} / (\text{trunc}((\text{pageuse} / 10) / \text{remsize}) + 1)$
Less than .33	$\text{partrempages} = \text{rows} / (\text{trunc}((\text{pageuse} / 3) / \text{remsize}) + 1)$
.33 or larger	$\text{partrempages} = \text{rows}$

- d. Add up the total number of pages with the following formula:

$$\text{tablesize} = \text{homepages} + \text{fullrempages} + \text{partrempages}$$

### Estimating Tables with Variable-Length Rows

When a table contains one or more VARCHAR, NVARCHAR, or LVARCHAR columns, its rows can have varying lengths. These varying lengths introduce uncertainty into the calculations. Form an estimate of the typical size of each VARCHAR or LVARCHAR column, based on your understanding of the data, and use that value when you make your estimates.

**Important:** When Universal Server allocates space to rows of varying size, it considers a page to be full when no room exists for an additional row of the maximum size.



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

- The maximum size of the table, which you calculate based on the maximum width allowed for all VARCHAR, NVARCHAR or LVARCHAR columns
- The projected size of the table, which you calculate based on a typical width for each VARCHAR, NVARCHAR, or LVARCHAR column

**To estimate the maximum number of the data pages**

1. To calculate *rowsize*, add together the maximum value for all column widths.
2. Use this value for *rowsize* and perform the calculations described in “Estimating Tables with Fixed-Length Rows” on page 3-9. The resulting value is called *maxsize*.

**To estimate the projected number of data pages**

1. To calculate *rowsize*, add together typical values for each of your variable-width columns. Informix suggests using the most-frequently occurring width within a column as the typical width for that column. If you do not have access to the data or do not wish to tabulate widths, you might choose to use some fractional portion of the maximum width, such as two-thirds (.67).
2. Use this value for *rowsize* and perform the calculations described in “Estimating Tables with Fixed-Length Rows” on page 3-9. The resulting value is called *projsize*.

*Selecting an Intermediate Value for the Size of the Table*

The actual table size should fall somewhere between *projsize* and *maxsize*. Based on your knowledge of the data, choose a value within that range that seems most reasonable to you. The less familiar you are with the data, the more conservative (higher) your estimate should be.



## Estimating Index Pages

The index pages associated with a table can add significantly to your space requirements. However, you might realize performance benefits when you define indexes. For more information on when an index might improve the response time for a query, refer to “Improving Query Performance with Indexes” on page 4-34.

When Universal Server creates an index on a user table, the index resides in its own tblspace regardless of whether it is fragmented or not fragmented. The data pages of the table remain in a separate tblspace from the index.

Universal Server uses the following types of indexes:

- B-tree
- R-tree
- Index that DataBlade modules provide

### B-Tree Indexes

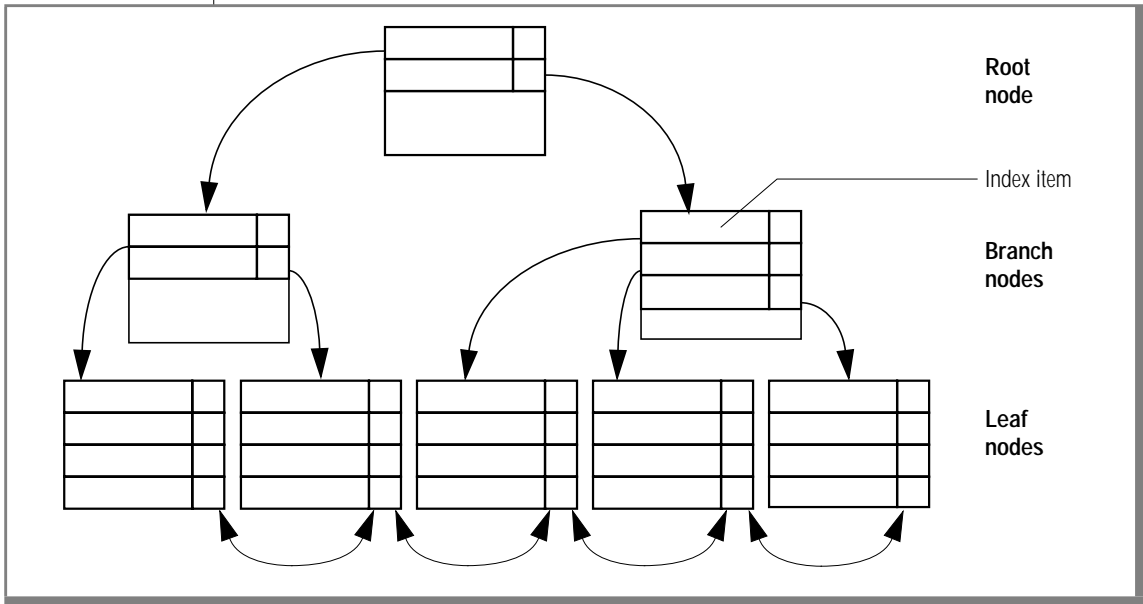
Universal Server uses a B-tree index for the following values:

- Columns that contain built-in data types (referred to as a *traditional B-tree index*)  
Built-in data types include character, datetime, integer, float, and so on. For more information on user-defined data types, refer to the *Informix Guide to SQL: Tutorial and Extending INFORMIX-Universal Server: Data Types*.
- Columns that contain one-dimensional user-defined data types (referred to as a *generic B-tree index*)  
User-defined data types include opaque and distinct data types. For more information on user-defined data types, refer to the *Informix Guide to SQL: Tutorial*.
- Values that a user-defined function returns (referred to as a *functional index*)

The return value can be a built-in or user-defined data type but not a large-object type (BYTE or TEXT data type). For more information on how to use functional indexes, refer to “Using a Functional Index” on page 4-54.

Figure 3-4 shows that a B-tree index is arranged as a hierarchy of pages.

**Figure 3-4**  
*B-Tree Structure of an Index*



Each node serves a different function. The following sections describe each node and the role that it plays in indexing:

- The topmost level of the hierarchy contains a single *root page*.
- Intermediate levels, when needed, contain *branch pages*. Each branch page contains entries that refer to a subset of pages in the next level of the index.
- The bottom level of the index contains a set of *leaf pages*. Each leaf page contains a list of index entries that refer to rows in the table.

A B-tree index uses a balanced tree structure for efficient record retrieval. This index is balanced when the leaf nodes are all at the same level from the root node. Each leaf page contains pointers to the next leaf node to the right and to the left. These pointers allow the database server to perform a range search without traversing to the next higher level and back down to the leaf pages.

The number of levels needed to hold an index depends on the number of unique keys in the index and the number of index entries that each page can hold. The number of entries per page depends, in turn, on the size of the columns to be indexed and the fill factor.

If the index page for a given table can hold 100 keys, a table of up to 100 rows might fit in a single index level (the root page), depending on the fill factor. When this table grows beyond 100 rows, to a size between 101 and 10,000 rows, it requires at least a two-level index (a root page and between 2 and 100 leaf pages). When the table grows beyond 10,000 rows, to a size between 10,001 and 1,000,000 rows, it requires at least a three-level index: the root page, a set of 100 branch pages, and a set of up to 10,000 leaf pages.

Universal Server uses the fill-factor value to determine how full to build each index page. You specify the fill-factor value in the FILLFACTOR configuration parameter. To override this ONCONFIG file value, you can specify the FILLFACTOR keyword in the CREATE INDEX statement. For more information on the FILLFACTOR configuration parameter, refer to the *INFORMIX-Universal Server Administrator's Guide*. For more information on the CREATE INDEX statement, refer to the *Informix Guide to SQL: Syntax*.

An index entry in a root or branch page consists of a key and a pointer to a lower-level (branch or leaf) page in the index. The key is the highest-value key contained within the lower-level page. B-tree indexes store key values in ascending or descending order.

An index entry in a leaf page consists of the following components:

- A key, which is a copy of one of the following values:
  - The indexed columns from one row of data
  - The returned value of a user-defined function on columns from one row of data
  - A combination of columns and returned values of a user-defined function on columns from one row of data

- One or more *row pointers*

A row pointer provides an address used to locate a row that contains the key. A unique index contains one index entry for every row in the table.

- A *delete flag* for each row that contains the key

The delete flag indicates whether a row with that particular key has been deleted.

- A *fragment ID*, if you create the index with an explicit fragmentation strategy on fragmented tables

The fragment ID, if present, identifies the table fragment (also referred to as a *partition*) in which the row resides.

**To estimate the number of index pages**

1. For each of the columns in the key, determine its length in the key with one of these formulas, depending on the data type of the column:
  - a. If the column is a built-in fixed-length data type, use the following formula:
$$\text{length} = \text{colsize}$$
  - b. If the column is a built-in variable-length data type, use the following formula:
$$\text{length} = \text{colsize} + 1$$
  - c. If the column contains fixed-length user-defined data types, use the following formula:
$$\text{length} = \text{colsize} + (\text{alignment})$$
  - d. If the column contains variable-length user-defined data types, use the following formula:
$$\text{length} = \text{colsize} + (\text{alignment} + 2)$$

To obtain *alignment*, select the value in the **align** column for the user-defined data type in the **sysxdtypes** system catalog table.
2. Determine the total key size based on the lengths calculated in the previous step.

$$\text{keysize} = \text{totallength} + 4$$

To obtain *totallength*, add up the length values in the previous step for each column in the key.

3. Calculate the expected proportion of unique entries to the total number of rows. This value is referred to as *propunique*. If the index is unique or very few duplicate values are present, use 1 for *propunique*. If a significant proportion of duplicate entries exist, divide the number of unique index entries by the number of rows in the table to obtain a fractional value for *propunique*. If the resulting value for *propunique* is less than .01, use .01 in the calculations that follow.
4. Estimate the size of a typical index entry with one of the following formulas, depending on whether the table is fragmented or not and whether the index is explicitly fragmented or not:
  - a. For a nonfragmented table and a fragmented table with an index created without an explicit fragmentation strategy, use the following formula:
 
$$\text{entrysize} = \text{keysize} * \text{propunique} + 5$$
  - b. For fragmented tables with the index explicitly fragmented, use the following formula:
 
$$\text{entrysize} = \text{keysize} * \text{propunique} + 9$$
5. Determine the page size of the Universal Server system that you use. To retrieve the page size, run the following command:
 

```
oncheck -pr
```
6. Estimate the number of entries per index page with the following formula:
 
$$\text{pagents} = \text{trunc}(\text{pagefree} / \text{entrysize})$$

*pagefree* is the page size minus the page header (2,020 for a 2-kilobyte page size).

The **trunc()** function notation indicates that you should round down to the nearest integer value.

7. Estimate the number of leaf pages with the following formula:
 
$$\text{leaves} = \text{ceiling}(\text{rows} / \text{pagents})$$

The **ceiling()** function notation indicates that you should round up to the nearest integer value; *rows* is the number of rows that you expect to be in the table.

8. Estimate the number of branch pages at the second level of the index with the following formula:
 
$$\text{branches}_0 = \text{ceiling}(\text{leaves} / \text{pagents})$$

9. If the value of `branches0` is greater than 1, more levels remain in the index. To calculate the number of pages contained in the next level of the index, use the following formula:

$$\text{branches}_{n+1} = \text{ceiling}(\text{branches}_n / \text{pages})$$

`branchesn` is the number of branches for the last index level you calculated.

`branchesn+1` is the number of branches in the next level.

10. Repeat the calculation in step 9 for each level of the index until the value of `branchesn+1` equals 1.
11. Add up the total number of pages for all branch levels calculated in steps 8 through 10. This sum is called the *branchtotal*.
12. Use the following formula to calculate the number of pages in the compact index:

$$\text{compactpages} = (\text{leaves} + \text{branchtotal})$$

13. If you use a fill factor less than 100 for indexes, the size of the index increases. The fill factor for indexes is set with the `FILLFACTOR` parameter in the `ONCONFIG` file or the `FILLFACTOR` keyword in the `CREATE INDEX` statement. To incorporate the fill factor into your estimate for index pages, use the following formula:

$$\text{indexpages} = 100 * \text{compactpages} / \text{FILLFACTOR}$$

Universal Server uses the value of the `FILLFACTOR` configuration parameter only when it builds index pages for rows that exist in the table at the time the index is created. The `FILLFACTOR` parameter or keyword does not apply to rows that are inserted or loaded after the index is created.

When you insert or load rows after the index is created, the resulting index can be filled anywhere between 50 and 100 percent, depending on the insertion order of the keys. Use **oncheck -pT** to see how much free space exists in the index pages.

As rows are deleted and new ones are inserted, the number of index entries can vary within a page. This method to estimating index pages yields a conservative (high) estimate for most indexes. For a more precise value, build a large test index using real data, and check its size with the **oncheck -pT** command.

*R-Tree Indexes*

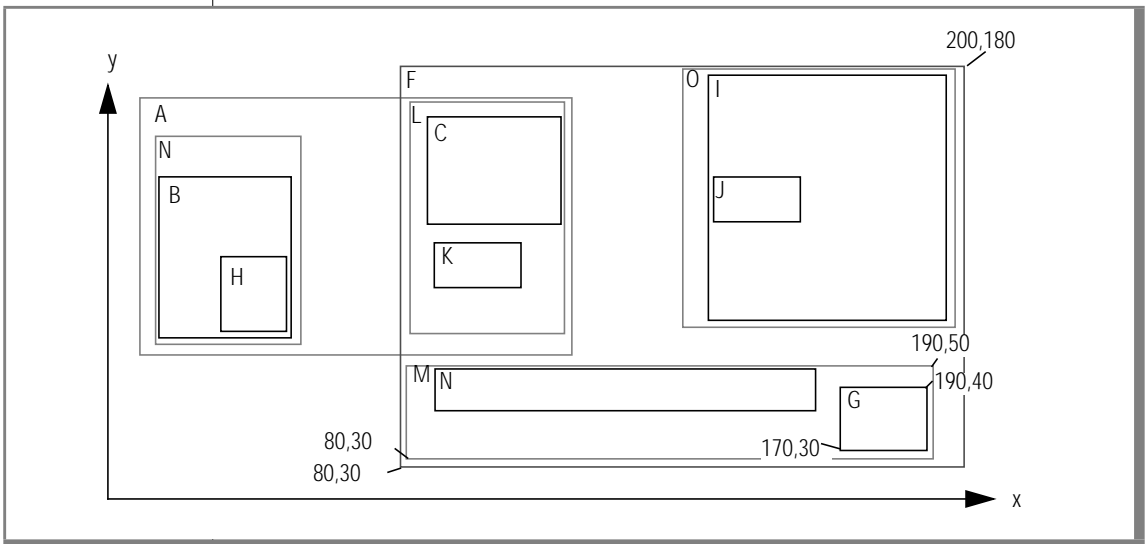
Universal Server uses an R-tree index for spatial data (two-dimensional, three-dimensional, and so forth).

R-trees use a concept called a *bounding box*, which is a set of coordinates that contain one or more objects. Figure 3-5 shows a set of objects (depicted by the solid lines) and their corresponding bounding boxes (depicted by the dotted lines). For example, bounding box M contains object G and object N. Bounding box F contains bounding box M.

Bounding boxes can grow in size as new objects are added.

An object can theoretically belong to more than one bounding box. For example, object C and object K can belong to either bounding box A or bounding box F. When you decide in which bounding box to place an object, the database server attempts to find the bounding box that has to grow the smallest amount to accommodate the new object.

**Figure 3-5**  
*Bounding Box Example*



A bounding box can be a rectangle or any other object, such as a polygon. Figure 3-5 on page 3-19 shows examples of rectangular bounding boxes, where the rectangle is represented by just two coordinates, the lower left corner, and the upper right corner. In Figure 3-5 on page 3-19, bounding box F would be stored with the following coordinates:

(80,30,200,180)

Bounding box M would be stored with the following coordinates:

(80,30,190,50)

Object G (which is a rectangle but could be a polygon, circle, or any other spatial object) would be stored with the following coordinates:

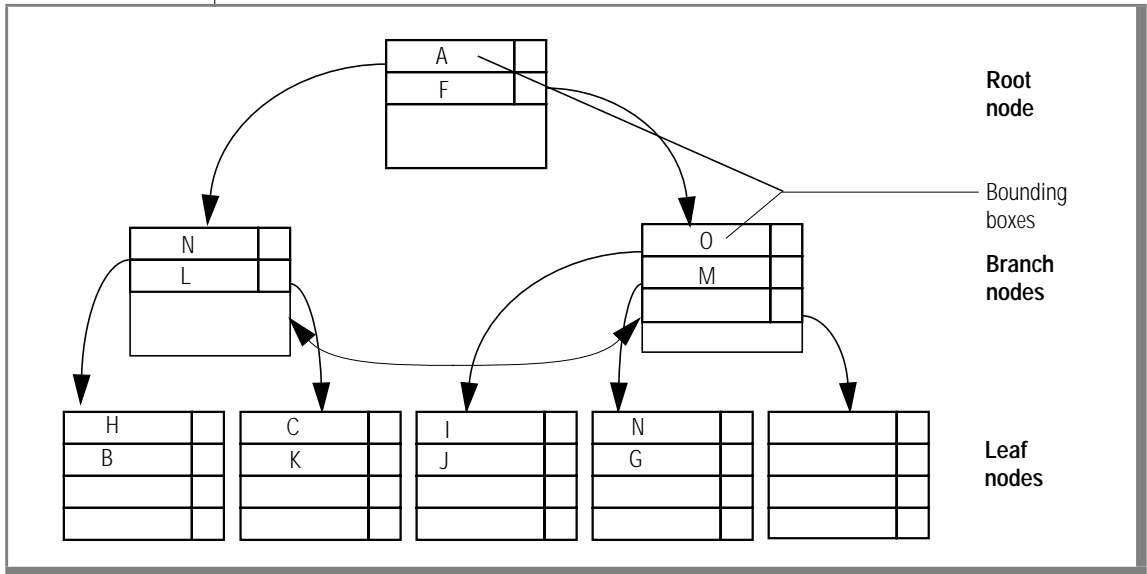
(170,30,190,40)

An R-tree index is arranged as a hierarchy of pages, similar to a B-tree index, as depicted in Figure 3-6 on page 3-21. The topmost level of the hierarchy contains a single *root page*. Intermediate levels, when needed, contain *branch pages*. Each branch page contains entries that refer to a subset of pages in the next level of the index. The bottom level of the index contains a set of *leaf pages*. Each leaf page contains a list of index entries that refer to rows in the table.



Figure 3-6 also shows how the objects and bounding boxes in Figure 3-5 on page 3-19 might be stored in an R-tree structure.

**Figure 3-6**  
R-Tree Structure



An index entry in a leaf page consists of the following information:

- A key that is the coordinates of the object
- A pointer to the row in the data page (also known as a row ID)
- A delete flag that marks the key for deletion
- The leaf tuple size

A row pointer provides an address used to locate a row that contains the key. (A unique index contains one index entry for every row in the table.) The delete flag indicates whether a row with that particular key has been deleted.

An index entry in a root or branch page consists of the following information:

- A bounding box that contains all the objects in its child nodes
- A pointer to a lower-level (branch or leaf) page in the index
- Overhead information

The number of levels needed to hold an index depends on the number of index entries that each page can hold. The number of entries per page depends, in turn, on the size of the key value.

When an index page fills, it splits into two pages, and a bounding box that contains the objects or bounding boxes in the newly created page is promoted to the higher page.

To estimate the number of index pages in an R-tree

1. Determine the size of the key value for the data type being indexed. This value is referred to as *colsize*.  
For example, key value sizes for the Spatial DataBlade module data types are shown in the **Size of Key Value** column in Figure 3-7. If you are indexing a user-defined spatial data type, use the size of the data type specified with the CREATE TYPE statement.
2. Determine the size of the bounding box for the data type, referred to as *boundingbox*. For example, bounding box sizes for the Spatial DataBlade module data types are shown in the **Size of Bounding Box** column in Figure 3-7.
3. Determine the size of each index entry in the leaf node, incorporating the overhead.

$$\text{entrysize} = \text{colsize} + 16$$

**Figure 3-7**  
*Spatial DataBlade Key Sizes*

Data Type	Size of Key Value	Size of Bounding Box
BOX	32 bytes	32 bytes
CIRC	56 bytes	56 bytes
POLY	Variable. Multiply the number of points in the average polygon stored in the table by 16 bytes.	48 bytes + (Size of Key Value)

4. Determine the page size of the Universal Server system that you use. To obtain the page size, run the following command:

```
oncheck -pr
```

5. Estimate the number of entries per index-leaf page with the following formulas:

$$\text{pagents} = \text{trunc}(\text{pagefree} / \text{entrysize})$$

$$\text{pagefree} = (\text{page size in bytes}) - (16 \text{ bytes for overhead}) - (\text{bounding box size})$$

For the bounding box size, see Figure 3-7 on page 3-22.

The **trunc()** function notation indicates that you should round down to the nearest integer value.

6. Estimate the number of leaf pages with the following formula:

$$\text{leaves} = \text{ceiling}(\text{rows} / \text{pagents})$$

The **ceiling()** function notation indicates that you should round up to the nearest integer value; *rows* is the number of rows that you expect to be in the table.

7. Determine the size of each entry in the branch node, incorporating the overhead of the bounding box. For example, the bounding box size for the Spatial DataBlade module data types is shown in the Size of Bounding column in Figure 3-7 on page 3-22.

$$\text{branchentrysize} = \text{boundingboxsize} + 8$$

8. Estimate the number of branch pages with the following formula:

$$\text{branches}_0 = \text{ceiling}(\text{leaves} / \text{pagents})$$

The **ceiling()** function notation indicates that you should round up to the nearest integer value.

9. If the value of  $\text{branches}_0$  is greater than 1, more levels remain in the index. To calculate the number of pages contained in the next level of the index, use the following formula:

$$\text{branches}_{n+1} = \text{ceiling}(\text{branches}_n / \text{pagents})$$

$\text{branches}_n$  is the number of branches for the last index level that you calculated.

$\text{branches}_{n+1}$  is the number of branches in the next level.

10. Repeat the calculation in step 9 for each level of the index until the value of  $\text{branches}_{n+1}$  equals 1.

11. Add up the total number of pages for all branch levels calculated in steps 8 through 10. This sum is called the *branchtotal*.
12. Use the following formula to calculate the number of pages in the index:

$$\text{pages} = (\text{leaves} + \text{branchtotal})$$

As rows are deleted and new ones are inserted, the number of index entries can vary within a page. This calculation yields an estimate for an R-tree that is very compact (leaf pages full). Your R-tree index might be larger depending on the activity within the table and the data that you store.

### *Indexes That DataBlade Modules Provide*

In Universal Server, users can access new data types that DataBlade modules provide. A DataBlade module can also provide a new index for the new data type. For example, the Excalibur Text Search DataBlade module provides an index to search text data. For more information, refer to the *Excalibur Text Search DataBlade Module User's Guide*.

For more information on the types of data and functions that each DataBlade module provides, refer to the user guide for each DataBlade module. For information on how to determine the types of indexes available in your database, see “Determining the Available Access Methods” on page 4-50.

### *Estimating Pages Occupied by Smart Large Objects*

In your estimate of the space required for a table, you should also consider the amount of sbspace storage for any smart large objects (BLOB and CLOB data types) that are part of the table. An sbspace contains user data areas and metadata areas. BLOB and CLOB data are stored in sbpages that reside in the user data area. The metadata area contains the smart-large-object attributes, such as sbpage size and whether or not the smart large object is logged. For more information on sbspaces, refer to the *INFORMIX-Universal Server Administrator's Guide*.

The first chunk of an sbspace must have a metadata area. When you add large objects and chunks to the sbspace after the initial allocation, Universal Server adds more control information into the metadata area. You might want to explicitly specify the size of the metadata area to ensure that the sbspace does not run out of metadata space.

### To estimate the size of the sbpace

1. Obtain the system-page size with **oncheck -pr**.
2. Estimate the size of the metadata area with one of the following methods:

- a. Let Universal Server automatically estimate the size of the metadata area for the first chunk in the sbpace.

To estimate the size of the metadata area, Universal Server uses the value that you specify (in kilobytes) with the **AVG\_LO\_SIZE** keyword in the **onspaces -Df** option.

- b. Use the following formula to calculate the total size of the metadata space in an sbpace:

$$\text{metatdata\_size} = (\text{pagesize} * ((\text{LOcount} * 1.2) + 50 + (\text{numchunks} * 28))) / 1024$$

*pagesize* is the system-page size in bytes from step 1.

*LOcount* is the number of smart large objects that you expect to reside in the sbpace.

*numchunks* is the number of chunks that you expect to reside in the sbpace.

You specify the metadata area size in kilobytes when you create the sbpace with the **onspaces -Ms** option.

3. Calculate the usable portion of the system page with the following formula:

$$\text{sbpuse} = \text{pagesize} - 32$$

4. Estimate the number of system pages required for a smart large object with one of the following methods:

- a. Use your estimated median size of the smart large objects in bytes (**AVG\_LO\_SIZE**) as follows:

$$\text{msbpages} = \text{ceiling}((\text{AVG\_LO\_SIZE} * 1024) / \text{sbpuse})$$

The **ceiling()** function notation indicates that you should round up to the nearest integer value.

- b. Alternatively, you can calculate the number of pages for each smart large object of *sbsize* *n*. This method is more precise but more time-consuming to calculate.

For each smart large object of *sbsize* *n*, calculate the number of pages that the smart large object occupies (*sbpages\_n*) with the following formula:

```
sbpages1 = ceiling(sbsize1/sbpuse)
sbpages2 = ceiling(sbsize2/sbpuse)
:
sbpages_n = ceiling(sbsize_n/sbpuse)
```

Calculate the average number of pages for smart large objects, as follows:

```
msbpages = (sbpages1 + sbpages2 + ... + sbpages_n)/n
```

5. Estimate the number of system pages for each smart-large-object user-data page with the following formula:

```
page_unit = msbpages / pagesize
```

You specify *page\_unit*, the number of system pages per sbpage, for the sbspace sbpage size when you create the sbspace with the **onspaces** command, as the following excerpt shows:

```
onspaces -c -S sblobsp4 -g page_unit ...
```

6. Estimate the user-data area (in kilobytes) of the sbspace with the following formula:

```
userdata_size = (LOcount * msbpages * pagesize) / 1024
```

*pagesize* is the system-page size in bytes from step 1.

*LOcount* is the number of smart large objects that you expect to reside in the sbspace.

*msbpages* is the number of system pages per smart large object.

7. Calculate the size of the sbspace with the following formula:

```
sbspacesize = metadata_size + userdata_size
```

You specify the sbspace size in kilobytes when you create the sbspace with the **onspaces** command, as the following excerpt shows:

```
onspaces -c -S sblobsp4 -g msbpages -s sbspacesize...
```

This method for estimating the size of an sbspace yields a conservative (high) estimate for most sbspaces. For a more precise value, create an sbspace, load real data, and check its fullness with the **oncheck -pS** command.

### ***Estimating Pages Occupied by Simple Large Objects***

In your estimate of the space required for a table, include blobpages for any simple large objects that are part of the table. The blobpages can reside in either the dbspace where the table resides or in a blobspace. For more information on when to use a blobspace, refer to “Locating Simple-Large-Object Data in the Tblspace or a Separate Blobspace” on page 3-28.

#### **To estimate the number of blobpages**

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

$$bpuse = pagesize - 32$$

2. For each simple large object of blobsize  $n$ , calculate the number of pages that the simple large object occupies ( $bpages\_n$ ) with the following formula:

$$\begin{aligned} bpages1 &= \text{ceiling}(blobsize1/bpuse) \\ bpages2 &= \text{ceiling}(blobsize2/bpuse) \\ &\vdots \\ bpages\_n &= \text{ceiling}(blobsize\_n/bpuse) \end{aligned}$$

The **ceiling()** function notation indicates that you should round up to the nearest integer value.

3. Add up the total number of pages for all simple large objects, as follows:

$$blobpages = bpages1 + bpages2 + \dots + bpages\_n$$

Alternatively, you can base your estimate on the median size of a simple large object; that is, the simple-large-object size that occurs most frequently. This method is less precise, but it is easier to calculate.

**To estimate the number of blobpages based on median simple-large-object size**

1. Calculate the number of pages required for a simple large object of median size as follows:

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

2. Multiply this amount by the total number of simple large objects, as follows:

```
blobpages = simple large object count * mpages
```

***Locating Simple-Large-Object Data in the Tblspace or a Separate Blobspace***

When you create a BYTE or TEXT column on magnetic disk, you can locate the column data in the tblspace or in a separate blobspace. You can often improve performance if you store simple-large-object data in a separate blobspace. (You can also store simple-large-object data on optical media, although this discussion does not apply to simple large objects stored in that way.) In the following example, a TEXT value is located in the tblspace, and a BYTE value is located in a blobspace named **rasters**:

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

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

When simple-large-object values are stored in the tblspace, the pages of their data are interspersed among the pages that contain rows, which can greatly increase the size of the table. When the database server reads only the rows and not the simple-large-object data, the disk arm must move farther than when the blobpages are stored apart. For example, the database server scans only the row pages when it performs a SELECT operation that does not retrieve a simple-large-object column and when it tests rows with a filter expression.



Another consideration is that disk I/O to and from a dbspace is buffered. Pages are stored in case they are needed again soon, and when pages are written, the requesting program can continue before the actual disk write takes place. However, because blob space data is expected to be voluminous, disk I/O to and from blob spaces is not buffered, and the program that makes the request is not allowed to proceed until all output is written to the blob space.

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

- When single data items are larger than one or two pages each
- When the number of pages of simple-large-object data is more than half the number of pages of row data

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

#### To separate row pages from blob pages

1. Load the entire table with rows in which the simple-large-object columns are null.
2. Create all indexes.  
The row pages and the index pages are now contiguous.
3. Update all the rows to install the simple-large-object data.  
The blob pages now appear after the pages of row and index data within the tblspace.

## Managing Indexes

An index is necessary on any column or composition of columns that must be unique. However, as discussed in Chapter 4, “Queries and the Query Optimizer,” the presence of an index can also allow the query optimizer to speed up a query. The optimizer can use an index in the following ways:

- To replace repeated sequential scans of a table with nonsequential access
- To avoid reading row data when processing expressions that name only indexed columns

- To avoid a sort (including building a temporary table) when executing the GROUP BY and ORDER BY clauses
- To avoid repeated executions of a function when the function is used in a filter (functional index)

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

The following sections discuss the following topics:

- Costs associated with indexes
- Maintaining indexes
- Dropping indexes
- Improving performance for index builds

### ***Costs Associated with Indexes***

The two costs associated with indexes are space and the time required to update.

#### *Space Costs of Indexes*

The first cost of an index is disk space. An estimating method is given in “Estimating Index Pages” on page 3-13. An index can add many pages to a tblspace. It is easy to have as many index pages as row pages in an indexed table.

The database server manages the space within an index efficiently. When an update transaction commits, the B-tree cleaner removes deleted index entries and, if necessary, rebalances the index nodes. This rebalancing helps reduce the amount of unused space within the index.

For more information on how Universal Server maintains an index tree, refer to the *INFORMIX-Universal Server Administrator's Guide*.

*Time Costs of Indexes*

The second cost of an index is time whenever the table is modified. The following descriptions assume that approximately two pages must be read to locate an index entry. That is the case when the index consists of a root page, one level of branch pages, and a set of leaf pages; the root page is presumed to be in a buffer already. The index for a very large table has at least two intermediate levels, so about three pages are read when referencing such an index.

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

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

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

The entry for the deleted row must be looked up (two or three pages in), and the leaf page must be rewritten. The write operation to update the index is performed in memory, and the leaf page is flushed when the LRU buffer that contains the modified page is cleaned. This operation requires two- or three-page accesses to read in the index pages if needed and one deferred-page access to write the modified page.

- When you insert a row, its entries must be inserted in all indexes.

A place in which to enter the inserted row must be found within each index (two or three pages in) and rewritten (one deferred page out), for a total of two or three immediate page accesses per index.

- When you update a row, its entries must be looked up in each index that applies to an altered column (two or three pages in).

The leaf page must be rewritten to eliminate the old entry (one deferred page out); then the new column value must be located in the same index (two or three more pages in) and the row entered (one more deferred page out).

Insertions and deletions change the number of entries on a leaf page. Universal Server requires some additional work to deal with a leaf page that has either filled up or emptied. However, if the number of index entries is greater than 100, this additional work occurs less than 1 percent of the time and can often be disregarded when making estimates of the I/O effect. For more information on what happens to a B-tree index when a row is deleted or inserted, refer to the *INFORMIX-Universal Server Administrator's Guide*.

In short, when a row is inserted or deleted at random, allow three to four added page I/O operations per index. When a row is updated, allow six to eight page I/O operations for each index that applies to an altered column. If a transaction is rolled back, all this work must be undone. For this reason, rolling back a transaction can take a long time.

Because the alteration of the row itself requires only two page I/O operations, index maintenance is the most time-consuming part of data modification. For one way to reduce this cost, refer to “Clustering” on page 4-39.

## Maintaining Indexes

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

When you know that a table is ordered by a certain index, the I/O is more efficient when the cluster index is used to scan the table. You can also be sure that when the table is searched on that column, it is read effectively in sequential order instead of nonsequentially. These points are covered in Chapter 4, “Queries and the Query Optimizer.”

In the **stores7** database, the **orders** table has an index, **zip\_ix**, on the zip code column. The following statement causes the database server to put the rows of the **customer** table in descending order by zip code:

```
ALTER INDEX zip_ix TO CLUSTER
```

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

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

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

Clustering is not preserved when you alter a table. When you insert new rows, they are stored physically at the end of the table regardless of their contents. When you update rows and change the value of the clustering column, the rows are written back into their original location in the table.

Clustering can be restored after the row order is disturbed by ongoing updates. The following statement reorders the table to restore data rows to the index sequence:

```
ALTER INDEX o_date_ix TO CLUSTER
```

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

Because clustering and reclustering take a lot of space and time, build the original table in the desired order.

## ***Dropping Indexes***

In some applications, most table updates can be confined to a single time period. You can set up your system so all updates are applied overnight or on specified dates.

When updates are performed as a batch, you can drop all nonunique indexes while you make updates, then create new indexes afterward. This strategy can have two good effects.

First, with fewer indexes to update, the updating program can run faster. Often, the total time to drop the indexes, update without them, and re-create them afterward is less than the time to update with the indexes in place. (For information on the time cost of updating indexes, refer to “Time Costs of Indexes” on page 3-31.)

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

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

Indexes also slow down the table population when you use the `LOAD` statement or the **dbload** utility. Loading a table that has no indexes is a quick process (little more than a disk-to-disk sequential copy), but updating indexes adds a great deal of overhead.

#### To load a table

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the nonunique indexes.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. You save time if the rows are presented in the correct sequence for at least one of the indexes (if you have a choice, make it the row with the largest key). This strategy minimizes the number of leaf pages that must be read and written.

### *Improving Performance for Index Builds*

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

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

1. Set PDQ priority to a value greater than 0.  
When you set PDQ priority to greater than 0, the index build can take advantage of the additional memory for parallel processing.
2. Do not set the **PSORT\_NPROC** environment variable.  
Informix recommends that you not set the **PSORT\_NPROCS** environment variable. If you have a computer with multiple CPUs, Universal Server uses two threads per sort when it sorts index keys and **PSORT\_NPROCS** is not set. The number of sorts depends on the number of fragments in the index, the number of keys, the key size, the values of PDQ priority, and the PDQ configuration parameters.
3. Allocate enough memory and temporary space to build the entire index.

Specify memory with **DS\_TOTAL\_MEMORY** and **DS\_MAX\_QUERIES**. For information on how to calculate sort memory, refer to the *INFORMIX-Universal Server Administrator's Guide*.

If not enough memory is available, use **onspaces -t** to create large temporary dbspaces and specify them in the **DBSPACETEMP** configuration parameter or environment variable. For information on how to optimize your temporary dbspaces, refer to the *INFORMIX-Universal Server Administrator's Guide*.

#### To estimate the amount of temporary space needed for an entire index build

1. Add up the total widths of the indexed column(s) or returned values from user-defined functions. This value is referred to as keysize.
2. Estimate the size of a typical item to sort with one of the following formulas, depending on whether the index is attached or not:
  - a. For a nonfragmented table and a fragmented table with an index created without an explicit fragmentation strategy
 
$$\text{sizeof\_sort\_item} = \text{keysize} + 4$$
  - b. For fragmented tables with the index explicitly fragmented
 
$$\text{sizeof\_sort\_item} = \text{keysize} + 8$$

3. Estimate the number of bytes needed to sort with the following formula:

```
temp_bytes = 2 * (rows * sizeof_sort_item)
```

This formula uses the factor 2 because everything is stored twice when intermediate sort runs use temporary space. Intermediate sort runs occur when not enough memory exists to perform the entire sort in memory.

The value for rows is the total number of rows that you expect to be in the table.

## Managing Extents

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

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

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

Because table sizes are not known, table space cannot be preallocated. Therefore, Universal Server adds extents only as they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when Universal Server creates a new extent that is adjacent to the previous extent, it treats both extents as a single extent.



### Choosing Table Extent Sizes

When you create a table, you can specify the size of the first extent as well as the size of the extents to be added as the table grows. The following example creates a table with a 512-kilobyte initial extent and 100-kilobyte added extents:

```
CREATE TABLE big_one (...column specifications...)
  IN big_space
  EXTENT SIZE 512
  NEXT SIZE 100
```

The default value for the extent size and the next-extent size is eight times the disk page size on your system. For example, if you have a 2-kilobyte page system, the default length is 16 kilobytes.

You can change the size of extents to be added with the ALTER TABLE statement. This change has no effect on extents that already exist. The following example changes the next-extent size of the table to 50 kilobytes:

```
ALTER TABLE big_one MODIFY NEXT SIZE 50
```

When you fragment an existing table, you might want to adjust your next-extent size because each fragment requires less space than the original, unfragmented table. If your unfragmented table was defined with a large next-extent size, Universal Server uses that same size for the next extent on *each* fragment, which results in over-allocation of disk space. Each fragment requires only a proportion of the space for the entire table.

For example, if you fragment the preceding **big\_one** sample table across five disks, you can alter the next-extent size to one-fifth the original size. For more information on the ALTER FRAGMENT statement, see the *Informix Guide to SQL: Syntax*. The following example changes the next-extent size to one-fifth of the original size:

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

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

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

When you assign an extent size to these kinds of tables, avoid creating large numbers of extents. A large number of extents causes the database server to spend extra time finding the data. In addition, an upper limit exists on the number of extents allowed. (For more information, refer to “Upper Limit on Extents” on page 3-40.)

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

#### To allocate space for table extents

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

As the dbspace fills up, what happens if you do not have enough contiguous space to create an extent of the size you specified? In that case, Universal Server allocates the largest contiguous extent that it can.

### ***Choosing Index Extent Sizes***

Universal Server bases the extent size of an index on the extent size for the corresponding table, regardless of whether the index is fragmented or not fragmented. Universal Server uses the ratio of the row size to index-key size to assign an appropriate extent size for the index, as the following formula shows:

$$\text{Index extent size} = (\text{index row size} / \text{table row size}) * \text{table extent size}$$

Universal Server also uses this same ratio for the next-extent size for the index:

$$\text{Index next extent size} = (\text{index row size} / \text{table row size}) * \text{table next extent size}$$



***Important:*** OnLine Dynamic Server supports attached indexes. An attached index resides in the same tblspace as the corresponding table. In an attached index, the index and data pages might be interleaved. Universal Server creates indexes that are known as detached indexes for OnLine Dynamic Server. A detached index resides in a separate tblspace from the corresponding user table.

### ***Upper Limit on Extents***

Do not allow a table to acquire a large number of extents because an upper limit exists on the number of extents allowed. When you try to add an extent after you reach the limit, you receive ISAM error -136 (No more extents) after an INSERT request.

The upper limit on extents depends on the page size and the table definition. To learn the upper limit on extents for a particular table, use the following set of formulas:

$$\begin{aligned}vcspace &= 8 * vcolumns + 136 \\tcspace &= 4 * tcolumns \\ixspace &= 12 * indexes \\ixparts &= 4 * icolumns \\extspace &= pagesize - (vcspace + tcspace + ixspace + ixparts + 84) \\maxextents &= extspace / 8\end{aligned}$$

***vcolumns*** is the number of columns that contain simple-large-object and VARCHAR data.

***tcolumns*** is the number of columns in the table.

***indexes*** is the number of indexes on the table.

***icolumns*** is the number of columns named in those indexes.

***pagesize*** is the size of a page reported by **oncheck -pr**.

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

To help stay within the limit:

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

### Checking for Extent Interleaving

When two or more growing tables share a dbspace, extents from one tblspace can be placed between extents from another tblspace. When this happens, the extents are said to be *interleaved*. Interleaving creates gaps between the extents of a table, as Figure 3-8 shows. Performance suffers when disk seeks for a table must span more than one extent, particularly for sequential scans. Try to optimize the table-extent sizes, which limits head movement. Also consider placing the tables in separate dbspaces.



**Figure 3-8**  
*Interleaved Table  
Extents*

Monitor Universal Server chunks periodically to check for extent interleaving. Execute **oncheck -pe** to obtain the physical layout of information in the chunk. The following information appears:

- Dbspace name and owner
- Number of chunks in the dbspace
- Sequential table layout and free space in each chunk
- Number of pages dedicated to each table extent or free space

This output is useful to determine the degree of extent interleaving. If Universal Server cannot allocate an extent in a chunk despite an adequate number of free pages, the chunk might be badly interleaved.

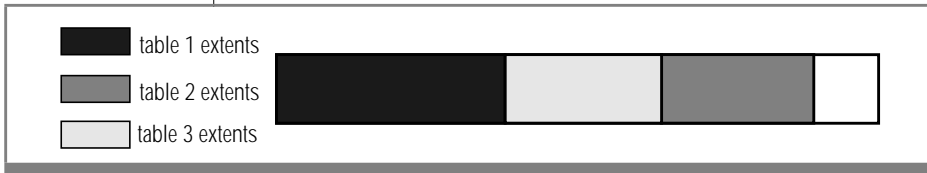
### Eliminating Interleaved Extents

You can eliminate interleaved extents with one of the following methods:

- Reorganize the tables with the UNLOAD and LOAD statements
- Create or alter an index to cluster
- Use the ALTER TABLE statement

### *Reorganizing Dbspaces and Tables to Eliminate Extent Interleaving*

You can rebuild a dbspace to eliminate interleaved extents, as Figure 3-9 illustrates. The order of the reorganized tables within the dbspace is not important, but the pages of each reorganized table should be together so that no lengthy seeks are required to read the table sequentially. When you read a table nonsequentially, the disk arm ranges over only the space occupied by that table.



**Figure 3-9**  
*A Dspace  
Reorganized to  
Eliminate  
Interleaved Extents*

#### **To reorganize tables in a dbspace**

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

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

You can also unload a table with the UNLOAD statement in DB-Access and reload the table with the companion LOAD statement or the **dbload** utility. However, these operations convert the table into character form. The **onload** and **onunload** utilities work with binary copies of disk pages.

You can reorganize a single table in two more ways. If you use the ALTER TABLE statement to add or drop a column or to change the data type of a column, the table is copied and reconstructed. If you create a clustered index or alter an index to cluster, the table is sorted and rewritten. (See “Clustering” on page 4-39.) In both cases, the table is written on other areas of the dbspace. However, if other tables are in the dbspace, no guarantee exists that all the new extents are adjacent.

*Creating or Altering an Index to Cluster*

Depending on the circumstances, you can eliminate extent interleaving if you create a clustered index or alter an index to cluster. When you use the TO CLUSTER clause of the CREATE INDEX or ALTER INDEX statement, Universal Server sorts and reconstructs the table. The TO CLUSTER clause reorders rows in the physical table to match the order in the index. For more information on clustering, refer to “Clustering” on page 4-39.

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

- The chunk must contain contiguous space that is large enough to rebuild each table.
- Universal Server must use this contiguous space to rebuild the table.  
If blocks of free space exist before this larger contiguous space, Universal Server might allocate the smaller blocks first. Universal Server allocates space for the ALTER INDEX process from the beginning of the chunk, and looks for blocks of free space that are greater than or equal to the size that is specified for the next extent. When Universal Server rebuilds the table with the smaller blocks of free space that are scattered throughout the chunk, it does not eliminate extent interleaving.

To display the location and size of the blocks of free space, execute the **oncheck -pe** command.

**To use the TO CLUSTER clause of the ALTER INDEX statement**

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

This step eliminates interleaving the extents when you rearrange the rows to rebuild the table.

3. Re-create all the other indexes.

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

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

To prevent the problem from recurring, consider increasing the size of the tblspace extents. For more information, see the *Informix Guide to SQL: Tutorial*.

### *Using ALTER TABLE to Eliminate Extent Interleaving*

If you use the ALTER TABLE statement to add or drop a column or to change the data type of a column, Universal Server copies and reconstructs the table. When Universal Server reconstructs the entire table, the table is rewritten to other areas of the dbspace. However, if other tables are in the dbspace, the new extents might not be adjacent to each other.



***Important:*** *If you only add one or more columns at the end of a table, Universal Server does not copy and reconstruct the table during the ALTER TABLE operation. In this case, Universal Server uses an in-place ALTER TABLE algorithm to modify each row when it is updated (rather than during the ALTER TABLE operation). For more information on the conditions and syntax when this in-place ALTER TABLE algorithm is used, refer to the “Informix Guide to SQL: Syntax.”*

### ***Reclaiming Unused Space Within an Extent***

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

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

- ALTER INDEX
- LOAD and UNLOAD
- ALTER FRAGMENT



*Reclaiming Space in an Empty Extent with ALTER INDEX*

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

For more information about the syntax of the ALTER INDEX statement, refer to the *Informix Guide to SQL: Syntax*. For more information on clustering, refer to “Clustering” on page 4-39.

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

If the table does not include an index, you can unload the table, re-create the table (either in the same dbspace or in another one), and reload the data with the UNLOAD and LOAD statements or the **onunload** and **onload** utilities. For further information about selecting the correct utility or statement to use, refer to the *Informix Migration Guide*.

*Releasing Space in an Empty Extent with ALTER FRAGMENT*

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

## Managing Smart Large Objects

One of the most important areas to monitor in an sbspace is the metadata page use. When you create an sbspace, you specify the size of the metadata area. Also, any time you add a chunk to the sbspace, you can specify that metadata space be added to the chunk.

If you attempt to insert a new smart large object but no metadata space is available, you receive an error. To prevent this problem, the administrator should monitor metadata space availability.

To monitor the amount of free metadata space, run the following utility:

```
oncheck -pS spacename
```

To add metadata space, allocate an additional chunk to the sbspace.

---

## Denormalizing the Data Model to Improve Performance

The entity-relationship data model described in the *Informix Guide to SQL: Tutorial* produces tables that contain no redundant or derived data and tables that are well structured by the tenets of relational theory.

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

### Using Shorter Rows for Faster Queries

Tables with shorter rows usually yield better performance than tables with longer rows because disk I/O is performed in pages, not in rows. The shorter the rows of a table, the more rows occur on a page. The more rows per page, the fewer I/O operations it takes to read the table sequentially, and the more likely it is that a nonsequential access can be satisfied from a buffer.

The entity-relationship data model puts all the attributes of one entity into a single table for that entity. For some entities, this strategy can produce rows of awkward lengths. To shorten the rows, place columns in separate tables that are associated by duplicate key values in each table. As the rows get shorter, query performance should improve.

## Expelling Long Strings

The most bulky attributes are often character strings. Removing them from the entity table makes the rows shorter. You can use the following methods to expel long strings:

- Use VARCHAR columns.
- Use TEXT simple large objects.
- Move strings to a companion table.
- Build a symbol table.

### *Using VARCHAR Strings*

A database might contain CHAR columns that can be converted to VARCHAR. You can use a VARCHAR column to shorten the average row when the average length of the text string in the CHAR column is at least two bytes shorter than the width of the column. For information about other character data types, refer to the *Guide to GLS Functionality*. ♦

VARCHAR data is immediately compatible with most existing programs, forms, and reports. You might need to recompile any forms produced by application development tools to recognize VARCHAR columns. Always test forms and reports on a sample database after you modify the table schema.

### *Using TEXT Simple Large Objects*

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

### *Moving Strings to a Companion Table*

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

### ***Building a Symbol Table***

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

For example, the **customer.city** column contains city names. Some city names are repeated down the column, and most rows have some trailing blanks in the field. Using the VARCHAR data type eliminates the blanks but not the duplication.

You can create a table named **cities**, as the following example shows:

```
CREATE TABLE cities (  
    city_num SERIAL PRIMARY KEY,  
    city_name VARCHAR(40) UNIQUE  
)
```

You can change the definition of the **customer** table so that its **city** column becomes a foreign key that references the **city\_num** column in the **cities** table.

Change any program that inserts a new row into **customer** to insert the city of the new customer into **cities**. The database server return code in the **SQLCODE** field of the SQL Communications Area (SQLCA) can indicate that the insert failed because of a duplicate key. It is not a logical error; it simply means that an existing customer is located in that city. For information about the SQLCA, refer to the *Informix Guide to SQL: Tutorial*.

Besides changing programs that insert data, you also must change all programs and stored queries that retrieve the city name. The programs and stored queries must use a join into the new **cities** table to obtain their data. The extra complexity in programs that insert rows and the extra complexity in some queries is the cost of giving up theoretical correctness in the data model. Before you make the change, be sure that it returns a reasonable savings in disk space or execution time.

### **Splitting Wide Tables**

Consider all the attributes of an entity that has rows that are too wide for good performance. Look for some theme or principle to divide them into two groups. Split the table into two tables, a primary table and a companion table, repeating the primary key in each one. The shorter rows allow each table to be queried or updated quickly.

### ***Division by Bulk***

One principle on which you can divide an entity table is bulk. Move the bulky attributes, which are usually character strings, to the companion table. Keep the numeric and other small attributes in the primary table. In the demonstration database, you can split the **ship\_instruct** column from the **orders** table. You can call the companion table **orders\_ship**. It has two columns, a primary key that is a copy of **orders.order\_num** and the original **ship\_instruct** column.

### ***Division by Frequency of Use***

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, they can be moved out to a companion table. In the demonstration database, it could be that the **ship\_instruct**, **ship\_weight**, and **ship\_charge** columns are queried only in one program. In that case, you can move them out to a companion table.

### ***Division by Frequency of Update***

Updates take longer than queries, and the updating programs lock index pages and rows of data during the update process, preventing querying programs from accessing the tables. If you can separate one table into two companion tables, one of which contains the most-updated entities and the other of which contains the most-queried entities, you can often improve overall response time.

### ***Costs of Companion Tables***

Splitting a table consumes extra disk space and adds complexity. Two copies of the primary key occur for each row, one copy in each table. Two primary-key indexes also exist. You can use the methods described in earlier sections to estimate the number of added pages.

Modify existing programs, reports, and forms that use **SELECT \*** because fewer columns are returned. Programs, reports, and forms that use attributes from both tables must perform a join to bring them together.

In this case, when you insert or delete a row, two tables are altered instead of one. If you do not coordinate the alteration of the two tables (by making them within a single transaction, for example), you lose semantic integrity.

## Redundant Data

Normalized tables contain no redundant data. Every attribute appears in only one table. Normalized tables also contain no derived data. Instead, data that can be computed from existing attributes is selected as an expression based on those attributes.

When you normalize tables, the amount of disk space used is minimized and makes updating the tables as easy as possible. However, normalized tables can force you to use joins and aggregate functions often, and that can be time consuming.

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

### *Adding Redundant Data*

A correct data model avoids redundancy by keeping any attribute only in the table for the entity that it describes. If the attribute data is needed in a different context, you make the connection by joining tables. But joining takes time. If a frequently used join affects performance, you can eliminate it by duplicating the joined data in another table.

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

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

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

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

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

---

## Table-Fragmentation Guidelines

This section discusses the performance considerations that are involved when you use table fragmentation. For an introduction to fragmentation concepts and methods, refer to the *INFORMIX-Universal Server Administrator's Guide*. For information about the SQL statements that manage fragments and perform parallel database queries, refer to the *Informix Guide to SQL: Syntax*.

One of the most frequent causes of poor performance in relational database systems is contention for data that resides on a single I/O device. Universal Server supports table fragmentation (also called *horizontal fragmentation* or *partitioning*), which allows you to distribute data from a single table onto multiple disk devices. Proper fragmentation of high-use tables can significantly reduce I/O contention.

Informix implements fragmentation by placing each fragment on a separate dbspace. Each dbspace that holds data from a fragmented table must be located on a separate disk to obtain the best results from table fragmentation.

The primary advantages of fragmentation are as follows:

- **Parallel scans.** Fragmentation allows Universal Server to read multiple fragments in parallel when large amounts of data must be read. Universal Server starts multiple scan threads to read the fragments in parallel.
- **Balanced I/O.** When a high degree of throughput is essential for on-line transaction processing (OLTP) performance, fragmentation allows you to balance I/O requests across disks to reduce contention and eliminate I/O bottlenecks.
- **Backup and restore.** Fragmentation provides for a finer granularity of backups and restores. A backup and restore can be performed at the dbspace level. Because a fragment resides in a dbspace, a backup and restore can be performed at the fragment level.
- **Higher availability.** You can specify whether to skip unavailable fragments when you scan or update a table. This feature is especially useful in large decision-support queries. Processing need not be interrupted if a portion of the data is unavailable.

When you fragment a table, the issues that pertain to tables described earlier in this chapter, such as placement on disk, extent sizing, and so forth, apply to individual table fragments. Because each fragment resides in its own dbspace, these issues must be addressed separately for each fragment. The following considerations differ between fragmented and nonfragmented tables:

- For fragmented tables, each fragment is placed in a separate, designated dbspace. For nonfragmented tables, the table can be placed in the default dbspace of the current database.
- For fragmented smart large objects, each fragment is placed in a separate, designated sbspace. For nonfragmented smart large objects, the table can be placed in the default sbspace.
- Table and index fragments are placed in separate tblspaces with their own extents and *tblspace IDs*. The tblspace ID is also known as the *fragment ID*.



- Extent sizes for a fragmented table are typically smaller than the extent sizes for a nonfragmented table because each extent holds data for only one fragment, not the entire table.
- In a fragmented table, the rowid is no longer a unique nonchanging pointer to the row on a disk. It has been replaced with the combination of fragment ID and rowid. These two fields are unique but can change over the life of the row. The fragment ID cannot be accessed in an application; the fragment ID/rowid combination is used only inside an index to point to the row.

Applications can still reference the rowid for a fragmented table. However, the rowid must be explicitly created as a column in fragmented tables in order to be used by an application.

- A fragmented index uses eight bytes of disk space per key value for the fragment ID/rowid combination. A nonfragmented index uses four bytes for the rowid.

Informix recommends that all tables have a primary key that can be used in an application to uniquely identify a row.

The performance of a fragmented table is governed primarily by the following factors:

- The fragmentation strategy that you use for allocating disk space to fragments
- The distribution scheme that you use for assigning rows to individual fragments

These factors are discussed in the following sections.

Decision-support queries typically create and access large temporary files; placement of temporary dbspaces is a critical factor for performance. For more information about placement of temporary files, refer to “Spreading Temporary Tables and Sort Files Across Multiple Disks” on page 3-7.

## **Considering Fragmentation Strategy**

When you formulate your fragmentation strategy, decide on the number of separate dbspaces across which to spread the table. When you fragment large tables, Informix recommends that you use a single chunk on a separate disk for each dbspace on which you place a fragment.

With individual queries, parallel I/O performance increases linearly with the number of fragments added. When a query accesses multiple smart large objects that are fragmented across multiple disks, the I/O can be executed in parallel. However, this increase in performance has a limit that is determined by the number of CPUs and the number of I/O ports.

Increasing the number of fragments can also reduce contention up to a point. However, as the percentage of rows in a fragment decreases, the likelihood that a search might span multiple fragments increases. At a certain point, the gains from fragmentation are offset by the need to coordinate multiple scan operations.

Before you decide on a fragmentation strategy, identify tables that are required by high-impact queries and determine what portions of each table these queries actually examine. Universal Server allows you to specify a data distribution that restricts the scope of queries to a subset of your table fragments.

When you eliminate fragments from a scan, you eliminate the associated I/O operations and delays. You also reduce demand for buffers and LRU queue activity. CPU cycles used to scan the fragment and manage the buffer can instead be used to perform calculations associated with the query or to support other queries or OLTP operations.

You can base your decision regarding the number of fragments on how you distribute your data, or you can distribute your data based on the number of available disks. However, examining the way in which your data is accessed can help you to recognize situations in which a particular distribution or a specific fragmentation strategy can improve performance.

For instance, if you know that a certain fragment of a table is accessed infrequently compared with other fragments, you might want to move this low-use fragment off your highest-speed disks. Alternatively, you could move the fragment to a chunk that does not contain centrally located disk cylinders if doing so would free enough prime space on disk for other high-use fragments or tables.

## Examining Your Data and Queries

To determine a fragmentation strategy, you must know how the data in a table is used. Take the following steps to gather information about a table that you contemplate fragmenting.

### To gather information about your table

1. Identify the queries that are critical to performance. Determine if the queries are OLTP or DSS.
2. Use the SET EXPLAIN statement to determine how the data is being accessed. (For details, refer to Chapter 4, “Queries and the Query Optimizer.”) Sometimes you can determine how the data is accessed by simply reviewing the SELECT statement along with the table schema.
3. Determine what portion of the data each query examines.  
For example, if certain rows in the table are read most of the time, you can isolate them into a small fragment to reduce I/O contention for other fragments.
4. Determine what statements create temporary files. Decision-support queries typically create and access large temporary files, and placement of temporary dbspaces can be critical to performance.
5. If particular tables are always joined together in a decision-support query, spread fragments for these tables across different disks.
6. Examine the columns in the table to determine which fragmentation scheme would keep each scan thread equally busy for the decision-support queries. (For more information on scan threads, refer to “Managing PDQ Resource Requirements and Queries” on page 4-64.) To see how the column values are distributed by creating a distribution on the column with the UPDATE STATISTICS statement and examining the distribution with **dbschema**, execute the following command:

```
dbschema -d database -hd table
```

### ***Fragmentation Goals***

Depending on your application and workload, try to strike a balance between the following fragmentation goals:

- Improved performance for individual queries
- Reduced contention between queries and transactions
- Increased data availability
- Finer granularity of backup and restore

The implications of each goal for your fragmentation strategy are discussed in the following sections.

### ***Improving Performance for Individual Queries***

When the primary goal of fragmentation is improved performance for individual queries, try to distribute all the rows of the table evenly over the different fragments. Overall query-completion time is reduced when Universal Server does not have to wait for data retrieval from a fragment that has more rows than other fragments.

When queries against a table perform sequential scans against significant portions of data, fragment the table rows only. Do not fragment the index. If an index is fragmented and a query has to cross a fragment boundary to access the index, the performance of the query can be worse than if you do not fragment.

When queries require sequential access and a fragmented index is present, you can set the OPTCOMPIND parameter to the value 2 to favor sequential scans.

When queries perform an index read to access data, you can fragment the indexes along with table rows to improve performance. If you determine that a fragmented index can improve performance and you use an expression-based distribution scheme as described in “Distribution Scheme” on page 3-60, use the same expression to fragment the index that you use to fragment the table.

If you use round-robin fragmentation, do not fragment your index. Consider placing that index in a separate dbspace from other table fragments.

Fragments from tables that are often joined should reside on separate disks.

### ***Reducing Contention Between Queries***

Fragmentation provides a means of reducing contention for data in tables that are used by multiple queries and OLTP applications. You can often use fragmentation to reduce contention when many simultaneous queries against a table perform index scans to return a few rows. For tables subject to this type of load, fragment both the index keys and data rows using an expression-based distribution scheme that allows each query to eliminate unneeded fragments from its scan.

To fragment a table for reduced contention, investigate which queries access which parts of the table. Next, fragment your data so that some of the queries are routed to one fragment while others access a different fragment.

Universal Server performs this routing when it evaluates the fragmentation rule for the table. Finally, store the fragments on separate devices.

Your success in reducing contention depends on how much you know about the distribution of data in the table and the scheduling of queries against the table. For example, if the distribution of queries against the table is set up so that all rows are accessed at roughly the same rate, try to distribute rows evenly across the fragments. However, if certain values are accessed at a higher rate than others, you can distribute the rows over the fragments unevenly to compensate. For more information, refer to “Designing an Expression-Based Distribution Scheme” on page 3-62.

### ***Increasing Availability of Data***

When you distribute table and index fragments across different disks or devices, you improve the availability of data during disk or device failures. Universal Server continues to allow access to fragments stored on disks or devices that remain operational. This availability has important implications for the following types of applications:

- Applications that do not require access to unavailable fragments  
A query that does not require Universal Server to access data in an unavailable fragment can still successfully retrieve data from fragments that are available. For example, if the distribution expression uses a single column (see “Designing an Expression-Based Distribution Scheme” on page 3-62), Universal Server can determine if a row is contained in a fragment without accessing the fragment. If the query accesses only rows that are contained in available fragments, a query can succeed even when some of the data in the table is unavailable.
- Applications that accept the unavailability of data  
Some applications might be designed to accept the unavailability of data in a fragment and retrieve the data that is available. Before they execute a query, these applications can execute the SET DATASKIP statement to specify which fragments to skip. Alternatively, the Universal Server administrator can specify which fragments are unavailable with the **onspaces -f** option.

If your fragmentation goal is increased availability of data, fragment both table rows and index keys so that if a disk drive fails, some of the data is still available.

If applications must always be able to access a subset of your data, keep those rows together in the same mirrored dbspace.

### ***Increasing Granularity for Backup and Restore***

Universal Server and **ontape** (see Chapter 5, “Using ontape to Back Up and Restore Data”) use the dbspace as the unit for storage and recovery. You can define a dbspace that is as small as several rows from a table or as small as several key values from an index, which increases the granularity of backup and restore operations.

If your goal is finer granularity of backup and restore operations, you can fragment table data, table index, or both, depending on what you need to recover rapidly in the event of a disk failure.

Pay attention to the following considerations when you develop a fragmentation strategy for finer granularity of backup and restore operations:

- Keep the size of your fragments small to reduce recovery time if a fragment becomes corrupted.
- Carefully consider how to group your rows. If a device that contains a dbspace fails, all the rows contained in the fragment of that dbspace are inaccessible.

For more information concerning archive and restore, see Chapter 5, “Using ontape to Back Up and Restore Data.”

## Distribution Scheme

Once you decide whether to fragment table rows, index keys, or both, and you decide how the rows or keys should be distributed over fragments, you must also decide on a scheme to implement this distribution.

Universal Server supports the following distribution schemes:

- **Round-robin.** This type of fragmentation randomly places rows in fragments to distribute rows evenly among fragments.  
  
For INSERT statements, the database server uses a hash function on a random number to determine the fragment in which to place the row. For INSERT cursors, the database server places the first row in a random fragment, the second in the next fragment, and so on. If one of the fragments gets full, it is skipped.  
  
For smart large objects, the database server places the first object in the first sbpace specified in the CREATE TABLE FRAGMENT BY statement, the second object in the next sbpace, and so on.
- **Expression based.** This type of fragmentation puts rows that contain similar values in the same fragment. You specify a *fragmentation expression* that defines criteria for how to assign a set of rows to each fragment. You can specify a *remainder fragment* that holds all rows that match the criteria for any other fragment.  
  
You cannot use expression-based fragmentation for smart large objects.



**Choosing Between Round-Robin and Expression-Based Distribution Schemes**

Figure 3-10 compares round-robin and expression-based distribution schemes in three important areas.

**Figure 3-10**  
*Distribution-Scheme Comparisons*

Dist. Scheme	Ease of Data Balancing	Fragment Elimination	Data Skip
Round-robin	Automatic. In addition, Universal Server maintains data balancing over time.	Universal Server cannot eliminate fragments.	When you use the data-skip feature, you cannot determine if the integrity of the transaction is compromised. However, you can always insert into a table fragmented by round-robin.
Expression based	Requires knowledge of the data distribution.	Depends on the fragment expressions. If nonoverlapping expressions on a single column are used, Universal Server can eliminate fragments for queries that have either range or equality expressions.	When you use the data-skip feature, you can determine whether the integrity of a transaction has been compromised. You cannot insert rows if the appropriate fragment for those rows is down.

The decision whether to use the round-robin or an expression-based distribution scheme is usually straightforward. Round-robin provides the easiest and surest way of balancing data. However, with this distribution scheme, you have no information about the fragments in which rows are located. This situation makes fragment elimination unworkable. Therefore, round-robin is generally only the correct choice when *all* the following conditions apply:

- Your queries tend to scan the entire table.
- You *do not* know the distribution of data to be added.
- Your applications tend not to delete many rows. (If they do, load balancing could be degraded.)

If none of these conditions apply, an expression-based distribution scheme is more appropriate. If your application calls for numerous decision-support queries that scan the entire table and you know what the data distribution is, the best choice is probably to fragment the data using an expression-based distribution scheme.

### ***Designing an Expression-Based Distribution Scheme***

The first step in designing an expression-based distribution scheme is to determine the distribution of data in the table, particularly the distribution of values for the column on which you base the fragmentation expression. You can obtain this information by running the UPDATE STATISTICS statement for the table and then examining the distribution with the **dbschema** utility.

Once you know the data distribution, you can design a fragmentation rule that distributes data across fragments as required to meet your fragmentation goal. If your primary goal is to improve performance, your fragment expression should generate an even distribution of rows across fragments.

If your primary fragmentation goal is improved concurrency, analyze the queries that access the table. If certain rows are accessed at a higher rate than others, you can compensate for this situation by opting for an uneven distribution of data over the fragments you create.

Avoid using columns that are subject to frequent updates in the distribution expression. Such updates can cause rows to move from one fragment to another (that is, be deleted from one and added to another), and this activity increases CPU and I/O overhead.

Try to create nonoverlapping regions based on a single column with no REMAINDER fragment for the best fragment-elimination characteristics. Universal Server eliminates fragments from query plans whenever the query optimizer can determine that the values selected by the WHERE clause do not reside on those fragments, based on the expression-based fragmentation rule by which you assign rows to fragments. Fragment elimination improves both response time for a given query and concurrency between queries. Because Universal Server does not need to read in unnecessary fragments, I/O for a query is reduced. Activity in the LRU queues is also reduced.

Figure 3-11 summarizes the fragment-elimination behavior for different combinations of expression-based distribution schemes and query expressions.

**Figure 3-11**  
*Fragment Elimination for Different Categories of Expression-Based  
Distribution Schemes and Query Expressions*

Type of Query (WHERE Clause) Expression	Type of Expression-Based Distribution Scheme		
	Nonoverlapping Fragments on a Single Column	Overlapping or Non- contiguous Fragments on a Single Column	Nonoverlapping Fragments on Multiple Columns
Range expression	Fragments can be eliminated.	Fragments cannot be eliminated.	Fragments cannot be eliminated.
Equality expression	Fragments can be eliminated.	Fragments can be eliminated.	Fragments can be eliminated.



**Important:** Where Figure 3-11 indicates that Universal Server can eliminate fragments, the effectiveness of fragment elimination is determined by the WHERE clause of the query in question. For example, consider a table fragmented with the following expression:

```

:
:
:
FRAGMENT BY EXPRESSION
100 < column_a AND column_b < 0 IN dbbsp1,
100 >= column_a AND column_b < 0 IN dbbsp2,
column_b >= 0 IN dbbsp3

```

Universal Server cannot eliminate any fragments from the search if your WHERE clause has the following expression:

```
column_a = 5 OR column_b = -50
```

However, if your WHERE clause has the following expression, Universal Server can eliminate the last fragment:

```
column_b = -50
```

Furthermore, if the WHERE clause has the following expression, Universal Server can eliminate the last two fragments:

```
column_a = 5 AND column_b = -50
```

## Monitoring Fragment Use

Once you determine a fragmentation strategy, you can monitor I/O activity to verify your strategy and determine whether I/O is balanced across fragments.

The **onstat -g ppf** command displays the number of read-and-write requests sent to each fragment that is currently open. Although these requests do not indicate how many individual disk I/O operations occur (a request can trigger multiple I/O operations), you can get a good idea of the I/O activity from these columns.

However, the output by itself does not show in which table a fragment is located. You can determine the table for the fragment by joining the **partnum** column in the output to the **partnum** column in the **sysfragments** table. The **sysfragments** table displays the associated **table id**. You can determine the table name for the fragment by joining the **table id** column in **sysfragments** to the **table id** column in **systables**.

## Fragmenting Indexes

When you fragment a table, the indexes associated with that table are fragmented implicitly, according to the fragmentation scheme you use. You can also use the **FRAGMENT BY** clause of the **CREATE INDEX** statement to fragment the index for any table explicitly. The fragmentation scheme for an index can differ from that of the table. You can fragment the index for any table by the expression distribution scheme.

If you do not want to fragment the index, you can put the entire index in a separate dbspace. You specify this separate dbspace in the **IN dbspace** clause when you create the index. For more information on the **CREATE INDEX** statement, refer to the *Informix Guide to SQL: Syntax*.

If Universal Server scans a fragmented index, multiple index fragments must be scanned and the results merged together. (The exception is if the index is fragmented according to some index-key range rule, and the scan does not cross a fragment boundary.) Because of this requirement, index scan performance might suffer if the index is fragmented.

Because of these performance considerations, Universal Server places the following restrictions on indexes:

- You cannot fragment indexes by round-robin.
- You cannot fragment unique indexes by an expression that contains columns that are not in the index key.

For example, the following statement is not allowed:

```
CREATE UNIQUE INDEX ia on tab1(col1)
  FRAGMENT BY EXPRESSION
    col2<10 in dbsp1,
    col2>=10 AND col2<100 in dbsp2,
    col2>100 in dbsp3;
```

## Improving Fragmentation

The following suggestions are guidelines on fragmenting tables and indexes:

- For optimal performance in decision-support queries, fragment the table to increase parallelism, but do not fragment the indexes. Detach the indexes, and place them in a separate dbspace.
- For best performance in OLTP, use fragmented indexes to reduce contention between sessions. You can often fragment an index by its key value, which means the OLTP query only has to look at one fragment to find the location of the row.

If the key value does not reduce contention, such as when every user looks at the same set of key values (for instance, a date range), consider fragmenting the index on another value used in the WHERE clause. To cut down on fragment administration, consider not fragmenting some indexes, especially if you cannot find a good fragmentation expression to reduce contention.

- Use round-robin fragmentation on data when the table is read sequentially by decision-support queries. Round-robin fragmentation is a good method for spreading data evenly across disks when no column in the table can be used for an expression-based fragmentation scheme. However, in most DSS queries, all fragments are read.

- If you are using expressions, create them so that I/O requests are balanced across disks. This does not necessarily mean an even distribution of data. If the majority of your queries access only a portion of data in the table, set up your fragmentation expression to spread active portions of the table across disks, even if this results in an uneven distribution of rows.
- Keep fragmentation expressions simple. Fragmentation expressions can be as complex as you wish. However, complex expressions take more time to evaluate and might prevent fragments from being eliminated from queries.
- Arrange fragmentation expressions so that the most-restrictive condition for each dbspace is tested within the expression first. When Universal Server tests a value against the criteria for a given fragment, evaluation stops when a condition for that fragment tests false. Thus, if the condition that is most likely to be false is placed first, fewer conditions need to be evaluated before Universal Server moves to the next fragment. For example, in the following expression, Universal Server tests all six of the inequality conditions when it attempts to insert a row with a value of 25:

```
x >= 1 and x <= 10 in dbspace1,  
x > 10 and x <= 20 in dbspace2,  
x > 20 and x <= 30 in dbspace3
```

By comparison, only four conditions in the following expression need to be tested: the first inequality for dbspace1 ( $x \leq 10$ ), the first for dbspace2 ( $x \leq 20$ ), and both conditions for dbspace3:

```
x <= 10 and x >= 1 in dbspace1,  
x <= 20 and x > 10 in dbspace2,  
x <= 30 and x > 20 in dbspace3
```

- Avoid any expression that requires a data type conversion. Type conversions increase the time to evaluate the expression. For instance, a DATE data type is implicitly converted to INTEGER for comparison purposes.
- Do not fragment on columns that change frequently unless you are willing to incur the administration costs. For example, if you fragment on a date column and older rows are deleted, the fragment with the oldest dates tends to empty, and the fragment with the recent dates tends to fill up. Eventually you have to drop the old fragment and add a new fragment for newer orders.

- Do not fragment every table. Identify the critical tables that are accessed most frequently. Put only one fragment for a table on a disk.
- Do not fragment small tables. Fragmenting a small table across many disks might not be worth the overhead of starting all the scan threads to access the fragments. Also, balance the number of fragments with the number of processors on your system.
- When you define a fragmentation strategy on an unfragmented table, check your next-extent size to ensure that you do not allocate large amounts of disk space for each fragment.





# Queries and the Query Optimizer

The Query Optimizer . . . . .	4-5
Factors That Affect Query-Plan Selection . . . . .	4-6
Updating Statistics . . . . .	4-7
Assessing Filters . . . . .	4-8
Assessing Indexes . . . . .	4-10
Selecting Table-Access Paths . . . . .	4-10
Displaying the Query Plan . . . . .	4-14
Time Costs of a Query . . . . .	4-15
Activities in Memory . . . . .	4-15
Time Used for a Sort . . . . .	4-16
The Cost of Reading a Row . . . . .	4-17
The Cost of Sequential Access . . . . .	4-18
The Cost of Nonsequential Access . . . . .	4-19
The Cost of Index Look-Ups . . . . .	4-19
The Cost of In-Place ALTER TABLE . . . . .	4-20
The Cost of Small Tables . . . . .	4-20
The Cost of Data Mismatches . . . . .	4-20
The Cost of GLS Functionality . . . . .	4-22
The Cost of Network Access . . . . .	4-22
The Importance of Table Order . . . . .	4-23
A Join Without Filters . . . . .	4-23
A Join with Column Filters . . . . .	4-25
Using Indexes . . . . .	4-26
The Sort-Merge Join Technique . . . . .	4-28
The Hash-Join Technique . . . . .	4-28
Improving Performance for a Particular Query . . . . .	4-29
Improving Filter Selectivity . . . . .	4-30
Avoiding Correlated Subqueries . . . . .	4-31
Avoiding Difficult Regular Expressions . . . . .	4-32
Avoiding Noninitial Substrings . . . . .	4-33

Creating Data Distributions . . . . .	4-33
Improving Query Performance with Indexes. . . . .	4-34
Filtered Columns . . . . .	4-36
Functions in Filters . . . . .	4-37
Join Expressions . . . . .	4-37
Order-By and Group-By Columns . . . . .	4-37
Avoiding Columns with Duplicate Keys . . . . .	4-38
Clustering . . . . .	4-39
Replacing Autoindexes with Permanent Indexes . . . . .	4-39
Using Composite Indexes . . . . .	4-39
Dropping and Rebuilding Indexes After Updates. . . . .	4-42
Reducing the Effect of Join and Sort Operations. . . . .	4-42
Avoiding or Simplifying Sort Operations. . . . .	4-42
Using Parallel Sorts . . . . .	4-43
Using Temporary Tables to Reduce Sorting Scope. . . . .	4-44
Improving Sequential Scans . . . . .	4-45
Eliminating Repeated Sequential Scans of Large Tables. . . . .	4-45
Using Unions to Eliminate Unneeded Scans. . . . .	4-46
Reviewing the Optimization Level . . . . .	4-47
Choosing an Index Type . . . . .	4-47
Defining Indexes for User-Defined Data Types . . . . .	4-48
The B-Tree Secondary Access Method. . . . .	4-49
Determining the Available Access Methods. . . . .	4-50
Using a User-Defined Secondary Access Method . . . . .	4-52
Using a Functional Index . . . . .	4-54
Using an Index Provided by a DataBlade Module. . . . .	4-56
Choosing Operator Classes for Indexes . . . . .	4-57
What Is an Operator Class?. . . . .	4-57
Strategy and Support Functions . . . . .	4-58
Default Operator Classes . . . . .	4-58
Built-In B-Tree Operator Class. . . . .	4-59
The B-Tree Strategy Functions . . . . .	4-59
The B-Tree Support Function . . . . .	4-60
Determining the Available Operator Classes . . . . .	4-61
Using an Operator Class. . . . .	4-62
Using an Existing Operator Class . . . . .	4-62
Using the R-Tree Index Operator Class . . . . .	4-63

Managing PDQ Resource Requirements and Queries. . . . .	4-64
How the Optimizer Structures a PDQ Query . . . . .	4-64
The Memory Grant Manager. . . . .	4-65
Allocating Resources for PDQ Queries . . . . .	4-67
Limiting the Priority of DSS Queries. . . . .	4-68
Adjusting the Amount of Memory . . . . .	4-72
Limiting the Number of Concurrent Scans. . . . .	4-73
Limiting the Maximum Number of Queries . . . . .	4-74
Managing Applications . . . . .	4-74
Using SET EXPLAIN . . . . .	4-74
Using OPTCOMPIND. . . . .	4-75
Using SET PDQPRIORITY . . . . .	4-75
User Control of Resources . . . . .	4-76
Universal Server Administrator Control of Resources . . . . .	4-76





**T**his chapter discusses the following topics:

- The *query optimizer*, which formulates a plan for fetching the data rows required to process a query
- Operations that take the most time when Universal Server processes a query
- Methods to improve the performance for a particular query
- Choosing an index type
- Choosing an operator class
- Parallel data query (PDQ) features that Universal Server supports

***Tip:** Taking advantage of the PDQ features in Universal Server provides the largest potential for improving performance of a query. PDQ provides the most substantial performance gains when you fragment your tables as described in Chapter 3, “Tables and Fragmentation,” and in the “INFORMIX-Universal Server Administrator’s Guide.”*

---

## The Query Optimizer

The query optimizer decides how to perform a query. A query might be performed in several ways. In the case of a single table, the optimizer must determine if an index should be used. In the case of joins, the optimizer must determine the join method (hash, sort-merge, or index), and the order in which tables are evaluated or joined.

A *query plan* is a specific way that a query might be performed. A query plan includes how to access the table(s), the order of joining tables, and how to use temporary tables. The query optimizer finds all feasible query plans. It assigns a cost to each component operation that is required under each plan and then selects the plan with the lowest cost for execution.

Among other things, the optimizer uses the following information to formulate a cost for each query plan:

- The number of I/O requests that are associated with each file-system access
- The CPU work that is required to determine which rows meet the query predicate
- The resources that are required to sort or group the data

When you use the UPDATE STATISTICS statement to maintain a few simple statistics about a table and its associated indexes, you provide the query optimizer with enough information to choose a query plan that minimizes the amount of time that is required to perform queries on that table.

## Factors That Affect Query-Plan Selection

The optimizer considers all possible query plans. It constructs all feasible plans simultaneously with a bottom-up, breadth-first search strategy. That is, the optimizer first constructs all possible join pairs and eliminates the more expensive of any *redundant* pairs. A redundant pair is a join pair that contains the same tables and produces the same set of rows as another join pair. For example, the join pair (A x B) is redundant with respect to (B x A) if neither produces an ordered set of rows; that is, the query does not request that output be ordered or grouped by columns in either table with the ORDER BY or GROUP BY clauses of the SELECT statement. Because either join produces the same rows, the more expensive of the two join pairs need not be considered.

If additional tables are present, the optimizer joins each remaining pair to a new table to form all possible join triplets and eliminates the more expensive of redundant triplets, and so on for each additional table to be joined. Once a nonredundant set of possible join combinations is generated, the optimizer selects the plan that appears to have the lowest cost for execution.

The quality of the query plan produced by the optimizer depends on the following items:

- Statistical information available to the optimizer
- Filters in the WHERE clause
- Availability of appropriate indexes
- OPTCOMPIND setting

### ***Updating Statistics***

Universal Server initializes a statistical profile of a table when that table is created. The query optimizer does not recalculate the profile for tables automatically. You instruct Universal Server to refresh this profile when you issue the UPDATE STATISTICS statement. In some cases, gathering the statistics could take longer than the query.

To ensure that the optimizer selects a query plan that best reflects the current state of your tables, issue the UPDATE STATISTICS statement for those tables on a periodic basis. For guidelines on how to execute the UPDATE STATISTICS statement, refer to “Creating Data Distributions” on page 4-33.

All Informix database servers provide the following information about tables:

- The number of rows in a table, as of the most recent UPDATE STATISTICS statement
- Whether a column is constrained to be unique
- The indexes that exist on a table, including the columns that they encompass, whether they are ascending or descending, and whether they are clustered
- The distribution of column values, when requested with the MEDIUM or HIGH keyword in the UPDATE STATISTICS statement

*Distributions* map the data values in a column into a defined number of bins. Each bin holds a percentage of data. If you create a distribution for a column associated with an index, the optimizer uses that distribution when it estimates the number of rows that match a query. For more information on distributions, refer to the *Informix Guide to SQL: Syntax*.

The system catalog tables that Universal Server maintains supply the following additional statistics:

- The number of disk pages occupied by row data
- The depth of the index B-tree structure (a measure of the amount of work that is needed to perform an index lookup)
- The number of disk pages that are occupied by index entries
- The number of unique entries in an index, which can be used to estimate the number of rows that are returned by an equality filter
- Second-largest and second-smallest key values in an indexed column

Only the second-largest and second-smallest key values are noted because the extreme values might have a special meaning that is unrelated to the rest of the data in the column. Universal Server assumes that key values are distributed evenly between the second largest and second smallest. Only the initial four bytes of these keys are stored.

For information on system catalog tables, refer to the *Informix Guide to SQL: Reference*.

### ***Assessing Filters***

The optimizer makes cost estimates based on the number of rows to be retrieved from each table in a query. In turn, the estimated number of rows is based on the *selectivity* of each conditional expression that is used within the WHERE clause. A conditional expression that is used to select rows is termed a *filter*. The selectivity is a value between 0 and 1 that indicates the proportion of rows within the table that the filter can pass. A selective filter, one that passes few rows, has a selectivity near 0; a filter that passes almost all rows has a selectivity near 1. For guidelines on filters, see “Improving Filter Selectivity” on page 4-30.



Universal Server calculates selectivities using data distributions. However, in the absence of data distributions, Universal Server calculates selectivities for the following filters based on current index information. The following table lists some of the selectivities that the optimizer assigns to filters of different types. This list is not exhaustive, and other expression forms might be added in the future.

Filter Expression	Selectivity (F)
<i>indexed-col = literal-value</i>	$F = 1 / (\text{number of distinct keys in index})$
<i>indexed-col = host-variable</i>	
<i>indexed-col IS NULL</i>	
<i>tab1.indexed-col = tab2.indexed-col</i>	$F = 1 / (\text{number of distinct keys in the larger index})$
<i>indexed-col &gt; literal-value</i>	$F = (2nd\text{-}max - \text{literal-value}) / (2nd\text{-}max - 2nd\text{-}min)$
<i>indexed-col &lt; literal-value</i>	$F = (\text{literal-value} - 2nd\text{-}min) / (2nd\text{-}max - 2nd\text{-}min)$
<i>any-col IS NULL</i>	$F = 1/10$
<i>any-col = any-expression</i>	
<i>any-col &gt; any-expression</i>	$F = 1/3$
<i>any-col &lt; any-expression</i>	
<i>any-col MATCHES any-expression</i>	$F = 1/5$
<i>any-col LIKE any-expression</i>	
EXISTS <i>subquery</i>	$F = 1$ if <i>subquery</i> estimated to return >0 rows, else 0
NOT <i>expression</i>	$F = 1 - F(\text{expression})$
<i>expr1</i> AND <i>expr2</i>	$F = F(\text{expr1}) \times F(\text{expr2})$
<i>expr1</i> OR <i>expr2</i>	$F = F(\text{expr1}) + F(\text{expr2}) - (F(\text{expr1}) \times F(\text{expr2}))$

(1 of 2)

Filter Expression	Selectivity (F)
<i>any-col</i> IN <i>list</i>	treated as <i>any-col</i> = <i>item</i> <sub>1</sub> OR...OR <i>any-col</i> = <i>item</i> <sub><i>n</i></sub>
<i>any-col relop</i> ANY <i>subquery</i>	treated as <i>any-col relop value</i> <sub>1</sub> OR ... OR <i>any-col relop value</i> <sub><i>n</i></sub> for estimated size of subquery <i>n</i>

**Key:**

*indexed-col*: first or only column in an index

*2nd-max, 2nd-min*: second-largest and second-smallest key values in indexed column

*any-col*: any column not covered by a preceding formula

(2 of 2)

**Assessing Indexes**

The database server requires indexes on columns that must be unique and are not specified as primary keys. If a column must contain unique values and the column is not specified as a primary key, you must create an index on that column. For more information on how indexes can improve the performance of a query, refer to “Improving Query Performance with Indexes” on page 4-34.

**Selecting Table-Access Paths**

The optimizer uses the filter selectivities and information about the physical layout of each index and table to estimate the cost of applying various access paths to retrieve the rows indicated by the query. To influence the access path chosen, set OPTCOMPIND.

*Single-Table Query*

For single-table scans, when OPTCOMPIND is set to 0, or when OPTCOMPIND is set to 1 and the current-transaction isolation level is Repeatable Read, the optimizer considers the following access paths:

- If an index is available, the optimizer considers whether to use it. In this case, the use of an index is called an *index scan*. If the columns requested are within a single index, the optimizer considers whether to execute the index scan without accessing the associated table. This scan is called a *key-only index scan*.
- If no index is available, the optimizer considers whether to scan the table in physical order without recourse to an index.

When OPTCOMPIND is not set, its value defaults to 2. When OPTCOMPIND is set to 2, or is set to 1 and the current isolation level is *not* Repeatable Read, the optimizer chooses the least-expensive method from among the following access paths:

- An index scan
- A key-only index scan
- A table scan

Cost comparison determines which method is less expensive. Universal Server derives cost from estimates of the number of I/O operations required, calculations to produce the results, rows accessed, sorting, and so forth.

*Multitable Query*

To influence the access path for join plans for a specific ordered pair of tables, set OPTCOMPIND.

Consider the following sample query that joins two tables, **t1** and **t2**. Table **t2** has an index defined on column **a**.

```
SELECT * FROM t1, t2
WHERE t1.a = t2.a;
```

The optimizer considers two join orderings: (t1, t2) and (t2, t1). As the optimizer considers each ordered join pair, it determines the best path for that specific pair. To determine if it can compare an existing index path with other possible paths for only that ordered join pair, the optimizer examines OPTCOMPIND. The optimizer compares the best path chosen for (t1, t2) with the best path chosen for (t2, t1) and selects the best of those two paths based on cost, not on OPTCOMPIND.

If OPTCOMPIND is set to 0, or is set to 1 and the current-transaction isolation level is Repeatable Read, the optimizer considers the following access paths for each ordered pair of tables:

- If an index already exists on the join column of the second table, the optimizer chooses to perform a *nested-loop join*.

A nested-loop join occurs when the first table is joined to the second table with an index on the second table. The index on the second table limits the number of rows that are examined in the second table for each row read in the first table. Therefore, for each row that is read in the first table, only the rows in the second table that match the index key value of the join column are examined.

For the ordered join pair (t1, t2) in the preceding example, table t2 has an index on column a. Therefore, the optimizer chooses a nested-loop join for this ordered pair.

- If no index is available, the optimizer chooses the least expensive of the following access paths:

- Hash join

One side of the join is used to construct a hash index, and the other side joins to it. A hash join is similar to a nested-loop join in which the permanent index is replaced by a hash index constructed at the time of execution.

- Dynamic-index join

A dynamic-index (also referred to as an *auto index*) join is a nested-loop join in which the index is constructed at the time of query execution.

- ❑ Sort-merge join

A sort-merge join is one in which each side of the join is sorted separately, and the result is then joined by merging the two sides.

For the ordered join pair (**t2**, **t1**), no index exists on **t1**. Therefore, the optimizer chooses the least expensive of these three access paths for this ordered pair.

When OPTCOMPIND is not set, is set to 2, or is set to 1 and the current isolation level is *not* Repeatable Read, the optimizer chooses the least-expensive access path from among those previously listed, and no preference is given to the nested-loop join.

Because of the nature of hash joins and sort-merge joins, an application with isolation mode set to Repeatable Read might temporarily lock all the records in tables that are involved in the join (including records that fail to qualify the join). This situation leads to decreased concurrency among connections. Conversely, nested-loop joins lock fewer records but provide inferior performance when a large number of rows is accessed. Thus, each join method offers advantages and disadvantages.

Generally, decision-support queries run faster if they use a Dirty Read isolation level. However, this isolation level is not suitable for queries that must return results that exactly reflect the current state of the database.

## Displaying the Query Plan

Any user who runs a query can display the query plan by executing the SET EXPLAIN ON statement before entering the SELECT statement for that query. The SET EXPLAIN ON statement writes an explanation of each query plan to a file for subsequent queries that user enters. Figure 4-1 shows a typical example.

**Figure 4-1**

*Output Produced by the SET EXPLAIN ON Statement*

```
QUERY:
-----
SELECT C.customer_num, O.order_num, SUM (I.total_price)
  FROM customer C, orders O, items I
  WHERE C.customer_num = O.customer_num
        AND O.order_num = I.order_num
  GROUP BY C.customer_num, O.order_num;

Estimated Cost: 102
Estimated # of Rows Returned: 12
Temporary Files Required For: GROUP BY

1) pubs.o: SEQUENTIAL SCAN

2) pubs.c: INDEX PATH

  (1) Index Keys: customer_num (Key-Only)
  Lower Index Filter: pubs.c.customer_num = pubs.o.customer_num

3) pubs.i: INDEX PATH

  (1) Index Keys: order_num
  Lower Index Filter: pubs.i.order_num = pubs.o.order_num
```

After it repeats the query, the optimizer shows its estimate of the work to be done (102 in Figure 4-1) in units used by the optimizer to compare plans. These units represent a relative time for query execution, with each unit assumed to be roughly equivalent to a typical disk access. The optimizer chooses this query plan because the estimated cost for its execution is the lowest among all the evaluated plans.

The optimizer also displays its estimate for the number of rows that the query is expected to produce.

In the body of the explanation, the optimizer lists the order in which tables are accessed and the method, or access path, that was used to read each table. The plan indicates that Universal Server is to perform the following actions:

1. Universal Server reads the **orders** table first. Because no filter exists on the **orders** table, Universal Server must read all rows. Reading the table in physical order is the least-expensive approach.
2. For each row of **orders**, Universal Server searches for matching rows in the **customer** table. The search uses the index on **customer\_num**. The notation *Key-Only* means that only the index need be read for the **customer** table because only the **c.customer\_num** column is used in the join and the output, and that column is an index key.

For each row of **orders** that has a matching **customer\_num**, Universal Server searches for a match in the **items** table with the index on **order\_num**.

---

## Time Costs of a Query

This section discusses the response-time effects of actions that Universal Server performs as it processes a query.

### Activities in Memory

Universal Server can process data only in memory. It must read rows into memory to evaluate those rows against the filters of a query. Once rows are found that satisfy those filters, Universal Server assembles the selected columns to prepare an output row in memory.

Most of these activities are performed quickly. Depending on the computer and its work load, Universal Server can perform hundreds or even thousands of comparisons each second. As a result, the time spent on in-memory work is usually a small part of the execution time.

Although some in-memory activities, such as sorting, take a significant amount of time, it takes much longer to read a row from disk than to examine a row that is already in memory.

## Time Used for a Sort

A sort requires in-memory work as well as disk work. The in-memory work depends on the number of columns being sorted, the width of the combined sort key, and the number of row combinations that pass the query filter. You can use the following formula to calculate the in-memory work that a sort operation requires:

$$W_m = (c + w) * N_{fr} \log_2(N_{fr})$$

$W_m$  is the in-memory work.

$c$  is the number of columns being ordered and represents the cost of extracting column values from the row and concatenating them into a sort key.

$w$  is proportional to the width of the combined sort key in bytes and stands for the work of copying or comparing one sort key. A numeric value for  $w$  depends strongly on the computer hardware in use.

$N_{fr}$  is the number of rows that pass the query filter.

When you sort, you write information temporarily to disk. You can direct that the disk writes occur in the operating-system file space or a dbspace managed by Universal Server.

The disk work depends on the number of disk pages where rows appear, the number of rows that meet the conditions of the query predicate, the number of rows that can be placed on a sorted page, and the number of merge operations that must be performed. You can use the following formula to calculate the disk work required by a sort operation:

$$W_d = p + (N_{fr} / N_{rp}) * 2 * (m - 1)$$

$W_d$  is the disk work.

$p$  is the number of disk pages.

$N_{fr}$  is the number of rows that pass the filters.

$N_{rp}$  is the number of rows that can be placed on a page.

$m$  represents the number of *levels of merge* that the sort must use.



The factor  $m$  depends on the number of sort keys that can be held in memory. If there are no filters, then  $N_{fr}/N_{rp}$  is equivalent to  $p$ .

When all the keys can be held in memory,  $m=1$  and the disk work is equivalent to  $p$ . In other words, the rows are read and sorted in memory.

For moderate- to large-sized tables, rows are sorted in batches that fit in memory, and then the batches are merged. When  $m=2$ , the rows are read, sorted, and written in batches. Then the batches are read again and merged, resulting in disk work proportional to the following value:

$$W_b = p + (2 * (N_{fr} / N_{rp}))$$

The more specific the filters, the fewer the rows that are sorted. As the number of rows increases and the amount of memory decreases, the amount of disk work increases.

To reduce the cost of sorting, use the following methods:

- Make your filters as specific (selective) as possible.
- Limit the projection list to the columns relevant to your problem.

## The Cost of Reading a Row

When Universal Server needs to examine a row that is not already in memory, it must read that row from disk. Universal Server does not read only one row; it reads the entire page (or pages) containing the row.

The actual cost of reading a page is variable and hard to predict. It is a combination of the following factors:

Buffering	The needed page might be in a page buffer already, so the cost of access is nearly zero.
Contention	If two or more applications require access to the disk hardware, I/O requests can be delayed.

Seek time	The slowest thing that a disk does is to <i>seek</i> ; that is, to move the access arm to the track that holds the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts. Seek time varies from zero to nearly a second.
Latency	The transfer cannot start until the beginning of the page rotates under the access arm. This <i>latency</i> , or rotational delay, depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from zero to a few milliseconds.

The time cost of reading a page can vary from microseconds for a page in a buffer, to a few milliseconds when contention is zero and the disk arm is already in position, to hundreds of milliseconds.

## **The Cost of Sequential Access**

Disk costs are lowest when Universal Server reads the rows of a table in physical order. When the first row on a page is requested, the disk page is read into a buffer page. Once the page is read in, it need not be read again; requests for subsequent rows on that page are filled from the buffer until all the rows on that page are processed. When one page is exhausted, the page for the next set of rows must be read in.

When you use raw devices for dbspaces and the table is organized properly, the disk pages of consecutive rows are placed in consecutive locations on the disk. This arrangement allows the access arm to move very little when it reads sequentially. In addition, latency costs are typically lowest when pages are read sequentially.

## The Cost of Nonsequential Access

Whenever a table is read in random order, additional disk accesses are required to bring in the needed rows as they are needed. Disk costs are higher when the rows of a table are read in a sequence unrelated to physical order on disk. Because the pages are not read sequentially from the disk, both seek and rotational delays occur before each page can be read. As a result, the disk-access time is much higher when a table is read nonsequentially than when that same table is read sequentially.

Nonsequential access often occurs when you use an index to locate rows. Although index entries are sequential, there is no guarantee that rows with adjacent index entries must reside on the same (or adjacent) data pages. In many cases, a separate disk access must be made to fetch the page for each row located through an index. If a table is larger than the page buffers, a page that contained a previously read row might be cleaned (removed from the buffer and written back to the disk) before a subsequent request for another row on that page can be processed. That page might have to be read in again.

Depending on the relative ordering of the table with respect to the index, you can sometimes retrieve pages that contain several needed rows. The degree to which the physical ordering of rows on disk corresponds to the order of entries in the index is called *clustering*. A highly clustered table is one in which the physical ordering on disk corresponds closely to the index.

## The Cost of Index Look-Ups

Universal Server encounters additional costs when it finds a row through an index. The index is stored on disk, and its pages must be read into memory along with the data pages that contain the desired rows.

An index look-up works down from the root page to a leaf page. (See “Managing Indexes” on page 3-29.) The root page, because it is used so often, is almost always found in a page buffer. The odds of finding a leaf page in a buffer depend on the size of the index, the form of the query, and the frequency of column-value duplication. If each value occurs only once in the index and the query is a join, each row to be joined requires a nonsequential look-up into the index, followed by a nonsequential access to the associated row in the table. However, if there are many duplicate rows per distinct index value, and the associated table is highly clustered, the added cost of joining through the index can be slight.

## The Cost of In-Place ALTER TABLE

When you add one or more columns to the end of a table, Universal Server uses an in-place alter algorithm to modify each row when it is updated (rather than during the alter table operation). After the alter table operation, Universal Server inserts rows with the latest definition. For more information on the conditions and syntax when the in-place alter table algorithm is used, refer to the *Informix Guide to SQL: Syntax*.

When Universal Server uses the in-place alter table algorithm, Universal Server locks the table for a short time because data rows are not converted during the alter operation. This action increases table availability and improves system throughput. However, if your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query because Universal Server reformats each row before it is returned.

## The Cost of Small Tables

A table is small if it occupies so few pages that it can be retained entirely in the page buffers. Operations on small tables are generally faster than operations on large tables.

As an example, in the **stores7** database, the **state** table that relates abbreviations to names of states has a total size less than 1,000 bytes; it fits in no more than two pages. You can include this table in any query at little cost. No matter how this table is used, it costs no more than two disk accesses to retrieve this table from disk the first time that it is required.

## The Cost of Data Mismatches

In general, an SQL statement does not encounter noticeable costs when the data type of a column used in a condition differs from the definition of the column in the CREATE TABLE statement.

For example, the following query contains a condition that compares a column to a data type value that differs from the table definition:

```
CREATE TABLE table1 (a integer, ...);
SELECT * FROM table1
WHERE a = '123';
```

Universal Server rewrites this query before execution to convert '123' to an integer. The **sqexplain.out** output shows the query in its adjusted format. This data conversion has no noticeable overhead.

To support data conversion between data types, Universal Server allows both implicit and explicit casts. An *implicit* cast refers to a system-implemented conversion between two data types. An *explicit* cast refers to a user-implemented conversion between two data types. The cast enables operations on different data types that would not otherwise be allowed. For more information on how to define casts for user-defined data types, refer to the *Informix Guide to SQL: Tutorial*.

The additional costs of a data mismatch are most severe when the query compares a character column with a noncharacter value and the length of the number is not equal to the length of the character column. For example, the following query contains a condition in the WHERE clause that equates a character column to an integer value because of missing quotes:

```
CREATE TABLE table2 (char_col char(3), ...);
SELECT * FROM table2
      WHERE char_col = 1;
```

This query finds all of the following values that are valid for **char\_col**:

```
' 1'
'001'
'1'
```

These values are not necessarily clustered together in the index keys. Therefore, the index does not provide a fast and correct way to obtain the data. The **sqexplain.out** output shows a sequential scan for this situation.

**Warning:** *Universal Server does not use an index when the SQL statement compares a character column with a noncharacter value that is not equal in length to the character column.*



## The Cost of GLS Functionality

When you sort and index certain data sets, it causes significant performance degradation. If you do not need a non-ASCII collation sequence, Informix recommends that you use the CHAR and VARCHAR data types for character columns whenever possible. Because CHAR and VARCHAR data require simple value-based comparison, sorting and indexing these columns is less expensive than for non-ASCII data types (NCHAR or NVARCHAR, for example). For more information on other character data types, see the *Guide to GLS Functionality*. ♦

## The Cost of Network Access

When you move data over a network, it imposes delays in addition to those you encounter with direct disk access. Network delays can occur when the application sends a query or update request across the network to a database server on another computer. Universal Server performs the query on the remote host computer; that database server returns the output rows to the application over the network.

Data sent over a network consists of command messages and buffer-sized blocks of row data. Although the details might differ depending on the network and the computers, Universal Server network activity follows a simple model in which one computer, the *sender*, sends a request to another computer, the *responder*. The responder replies with a block of data from a table.

Whenever data is exchanged over a network, delays are inevitable in the following situations:

- If the network is busy, the sender must wait its turn to transmit. Such delays are typically less than a millisecond. However, in a heavily loaded network, these delays can increase exponentially to tenths of seconds and more.
- The responder might be handling requests from more than one sender; when the request arrives, it might be queued for a time that can range from milliseconds to seconds.

- When the responder acts on the request, it incurs the time costs of disk access and in-memory operations that the preceding topics describe.
- Transmission of the response is again subject to network delays.

Network access is subject to extreme variability. In the best case, when neither the network nor the responder is busy, transmission and queueing delays are insignificant, and the responder sends a row almost as quickly as a local database server might. Furthermore, when the sender asks for a second row, the page is likely to remain in the page buffers of the responder.

Unfortunately, as network load increases, all these factors tend to worsen at the same time. Transmission delays rise in both directions, which increases the queue at the responder. The delay between requests decreases the likelihood of a page remaining in the page buffer of the responder. Thus, network-access costs can change suddenly and quite dramatically.

The optimizer that Universal Server uses assumes that access to a row over the network takes longer than access to a row in a local database. This estimate includes the cost of retrieving the row from disk and transmitting it across the network.

## **The Importance of Table Order**

The order in which tables are examined has an enormous effect on the speed of a join operation. This concept can be clarified with some examples that show how Universal Server works.

### ***A Join Without Filters***

The following SELECT statement calls for a three-way join:

```
SELECT C.customer_num, O.order_num
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
```

For this example, it is assumed that no index is available. Figure 4-2 shows a possible *query plan*, expressed in a programming pseudocode. A query plan states the order in which Universal Server examines tables and the methods by which it accesses the tables.

```
for each row in the customer table do:
  read the row into C
  for each row in the orders table do:
    read the row into O
    if O.customer_num = C.customer_num then
      for each row in the items table do:
        read the row into I
        if I.order_num = O.order_num then
          accept the row and send to user
        end if
      end for
    end if
  end for
end for
```

**Figure 4-2**  
*A Query Plan in Pseudocode*

This procedure reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for each row of the **customer** table
- All rows of the **items** table once for each row of the **customer-orders** pair

Other query plans are possible. Another plan merely reverses the roles of **customer** and **orders**: for each row of **orders**, it reads all rows of **customer** looking for a matching **customer\_num**. It reads the same number of rows in a different order and produces the same set of rows in different order. In this example, no difference exists in the amount of work that the two possible query plans need to do.



### A Join with Column Filters

The presence of a *column filter* changes things. A column filter is a WHERE expression that reduces the number of rows that a table contributes to a join. The following example shows the preceding query with a filter added:

```
SELECT C.customer_num, O.order_num
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
      AND O.paid_date IS NULL
```

The expression `O.paid_date IS NULL` filters out some rows, reducing the number of rows that are used from the **orders** table. Consider a plan that starts by reading from **orders**. Figure 4-3 displays it in pseudocode.

```
for each row in the orders table do:
  read the row into O
  if O.paid_date is null then
    for each row in the customer table do:
      read the row into C
      if O.customer_num = C.customer_num then
        for each row in the items table do:
          read the row into I
          if I.order_num = O.order_num then
            accept row and return to user
          end if
        end for
      end if
    end for
  end if
end for
```

**Figure 4-3**  
Filter Added to  
Query Plan

Let *pdnnull* represent the number of rows in **orders** that pass the filter. It is the value of `count(*)` that results from the following query:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

If you assume that one customer exists for every order, the plan in Figure 4-3 reads the following rows:

- All rows of the **orders** table once
- All rows of the **customer** table, *pdnnull* times
- All rows of the **items** table, *pdnnull* times

Figure 4-4 shows an alternative execution plan in pseudocode. It reads from the **customer** table first.

```
for each row in the customer table do:
  read the row into C
  for each row in the orders table do:
    read the row into O
    if O.paid_date is null and
       O.customer_num = C.customer_num then
      for each row in the items table do:
        read the row into I
        if I.order_num = O.order_num then
          accept row and return to user
        end if
      end for
    end if
  end for
end for
```

**Figure 4-4**  
Alternative Query  
Plan

Because the filter is not applied in the first step, which Figure 4-4 shows, this plan reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table, once for every row of **customer**
- All rows of the **items** table, *pnull* times

The query plans in Figure 4-3 on page 4-25 and Figure 4-4 produce the same output in a different sequence. They differ in that one reads a table *pnull* times and the other reads a table `SELECT COUNT(*) FROM customer` times. By choosing the appropriate plan, the optimizer can make a difference of thousands of disk accesses in a real application.

### Using Indexes

The preceding examples do not use indexes or constraints. The presence of indexes and constraints provides the optimizer with options that can drastically improve query-execution time. Figure 4-5 on page 4-27 shows in pseudocode the outline of a query plan for the previous query as it might be constructed with indexes.

```

for each row in the customer table do:
  read the row into C
  look up C.customer_num in index on orders.customer_num
  for each matching row in the orders index do:
    read the table row for O
    if O.paid_date is null then
      look up O.order_num in index on items.order_num
      for each matching row in the items index do:
        read the row for I
        construct output row and return to user
      end for
    end if
  end for
end for

```

**Figure 4-5**  
A Query Plan That  
Uses Indexes

The keys in an index are sorted so that when the first matching entry is found, any other rows with identical keys can be read without further searching because they are located in physically adjacent positions. This query plan reads only the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once (because each order is associated with only one customer)
- Only rows in the **items** table that match *pdnull* rows from the **customer-orders** pairs

This query plan achieves a huge reduction in effort compared with the plans without indexes. An inverse plan, reading **orders** first and looking up rows in the **customer** tables by its index, is also feasible by the same reasoning.

Using an index incurs an additional cost over reading the table sequentially. Each entry, or set of entries with the same value, must be located in the index, and then for each entry in the index a random access must be made to the table to read the associated row.

The physical order of a table also affects the cost of index use. To the degree that a table is ordered relative to an index, the overhead of accessing multiple table rows in index order is reduced. For example, if the **orders** table rows are physically ordered according to the customer number, multiple retrievals of orders for a given customer would proceed more rapidly than if the table were ordered randomly.

**GLS**

When GLS is enabled, indexes that are built on NCHAR or NVARCHAR columns are sorted with a country-specific comparison value. For example, the Spanish double-l character (ll) might be treated as a single unique character instead of a pair of l's. ♦

### ***The Sort-Merge Join Technique***

The sort-merge join provides a cost-effective alternative to constructing a temporary index for a nested-loop join. The optimizer considers this method only when the join is based on an equality filter and no index exists that can accomplish the join. If OPTCOMPIND is set to 2, or set to 1 and the current transaction isolation level is *not* Repeatable Read, the optimizer considers plans that replace existing index nested-loop joins with a sort-merge join. No index is required when you perform a sort-merge join. For more information on OPTCOMPIND and the different join methods, refer to “Selecting Table-Access Paths” on page 4-10.

The sort-merge join does not change the fundamental strategy of the optimizer; it provides a richer set of alternatives and alters the way that the ORDER BY and GROUP BY paths are analyzed. The path chosen by the optimizer appears in the output when the SET EXPLAIN ON statement is issued, as described in “Displaying the Query Plan” on page 4-14 and in the *Informix Guide to SQL: Syntax*.

### ***The Hash-Join Technique***

The optimizer considers a hash join whenever it considers a sort-merge join and automatically selects a hash join whenever doing so appears advantageous. A hash join is used when it can perform more rapidly than the other join methods or when the complexity of the join expression precludes the use of other join methods. No index is required when Universal Server performs a hash join.

Hash joins can provide a significant performance advantage over the other join methods, especially where the size of the join tables is large. Hash joins can improve performance even when a table has not been fragmented. Hash joins are faster than sort-merge joins, which sort both tables. Typically, when Universal Server uses a hash join, the larger of the two tables does not have to be sorted.

The hash table is created in the virtual portion of shared memory. If inadequate memory is available, the hash table overflows to one of the temporary dbspaces listed in **DBSPACETEMP** or **PSORT\_DBTEMP**.

---

## Improving Performance for a Particular Query

Generally, the following operations in a query are expensive. When you study the SET EXPLAIN output for a given query and examining your data model, you can often arrange to avoid these costly operations:

- Reading multiple rows from disk nonsequentially, either by ROWID (for nonfragmented tables) or through an index
- Reading rows over a network
- Sorting
- Performing certain difficult pattern matches on regular expressions
- Indexing and sorting NCHAR, NVARCHAR, and LVARCHAR columns
- Comparing character fields with integer values

You can often improve performance by adjusting your query and data model with the following goals in mind:

- Eliminating table fragments (refer to “Considering Fragmentation Strategy” on page 3-54)
- Improving the selectivity of filters
- Creating data distributions to improve optimizer performance
- Improving query performance with indexes
- Creating indexes for user-defined data types and functions
- Reducing the effect of join and sort operations
- Improving sequential scans
- Reviewing the optimization level



**Important:** When you test a high-impact query, isolate regular database operations from the effect of your tests. You can safeguard the performance of your production database server if you perform your tests on another server that resides on a separate host.

If you use a multiuser system or a network, where system load varies widely from hour to hour, you might need to perform your experiments at the same time each day to obtain repeatable results. Initiate tests when the system load is consistently light so that you are truly measuring the impact of your query only.

If the query is embedded in a complicated program, you can extract the SELECT statement and embed it in a DB-Access script.

If you are trying to improve performance of a large query, one that might take several minutes or hours to complete, you can prepare a scaled-down database in which your tests can complete more quickly. Using a scaled-down database can help you with testing, but you must be aware of the following potential problems:

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

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

## Improving Filter Selectivity

The greater the precision with which you specify the desired rows, the greater is the likelihood that your queries will complete quickly. You control the amount of information that the query evaluates using the WHERE clause of the SELECT statement. The conditional expression given in the WHERE clause is commonly called a *filter*.

The selectivity of a filter indicates the proportion of rows within the table that the filter can pass. Universal Server calculates selectivities with data distributions. However, in the absence of data distributions, Universal Server calculates default selectivities as shown in “Assessing Filters” on page 4-8.

To improve selectivities for filters, issue the `UPDATE STATISTICS` statement for your most frequently accessed tables periodically. For guidelines on how to execute the `UPDATE STATISTICS` statement, refer to “Creating Data Distributions” on page 4-33.

For best performance, avoid the following types of filters:

- Correlated subqueries
- Certain difficult regular expressions
- Noninitial substrings

These types of filters and the reasons to avoid them are described in the following sections.

### ***Avoiding Correlated Subqueries***

A subquery is correlated when the value that it produces depends on a value that is produced by the outer `SELECT`. Because the result of the subquery might be different for each row that Universal Server examines, the subquery begins anew for every row in the outer query if the current correlation values are different from the previous ones. The optimizer tries to use an index on the correlation values to cluster identical values together. This procedure can be extremely time consuming. Therefore, wherever possible, rewrite a correlated subquery as a join because it will execute at the same speed or faster (usually the case).

The following example is a correlated subquery:

```
SELECT item FROM a
WHERE item IN (SELECT item FROM b WHERE b.num = 50);
```

The following example shows the preceding correlated subquery rewritten as a join:

```
SELECT item FROM a, b
WHERE a.item = b.item AND b.num = 50;
```

When you see a subquery in a time-consuming `SELECT` statement, check if it is correlated. (An uncorrelated subquery, one in which no row values from the main query are tested within the subquery, is performed only once.) If it is correlated, try to rewrite the query as a join.

Often you can break such queries into separate queries for better performance. You can execute the subquery into a temporary table and join it to the table used in the outer-block query. For example, consider the following correlated subquery:

```
SELECT COMPANY FROM SALES S
      WHERE S.dollars_spent < (SELECT AVG(dollars_spent)
                               FROM SALES SQ
                               WHERE S.account_type = SQ.account_type);
```

The following statements show this correlated subquery rewritten as two separate queries. The first query executes the subquery into a temporary table, and the second query joins the temporary table to the outer-block query.

```
SELECT AVG(dollars_spent) avg_dollars_spent, account_type
FROM SALES S
GROUP BY account_type
INTO TEMP DOLLARS_BY_ACCOUNT;

SELECT COMPANY FROM SALES S, DOLLARS_BY_ACCOUNT DA
      WHERE S.dollars_spent < DA.avg_dollars_spent and
             S.account_type = DA.account_type;
```

### ***Avoiding Difficult Regular Expressions***

The MATCHES and LIKE keywords support *wildcard* matches, which are technically known as *regular expressions*. Some regular expressions are more difficult than others for Universal Server to process. A wildcard in the initial position, as in the following example (find customers whose first names do not end in y), forces Universal Server to examine every value in the column:

```
SELECT * FROM customer WHERE fname NOT LIKE '%y'
```

An index cannot be used with such a filter, so the table in this example must be accessed sequentially.

If a difficult test for a regular expression is essential, avoid combining it with a join. If necessary, process the single table and apply the test for a regular expression to select the desired rows. Save the result in a temporary table, and join that table to the others.

Regular-expression tests with wildcards in the middle or at the end of the operand do not prevent the use of an index when one exists.



### ***Avoiding Noninitial Substrings***

A filter that is based on a noninitial substring of a column also requires every value in the column to be tested, as the following example shows:

```
SELECT * FROM customer
WHERE zipcode[4,5] > '50'
```

An index cannot be used to evaluate such a filter.

The optimizer uses an index to process a filter that tests an initial substring of an indexed column. However, the presence of the substring test can interfere with the use of a composite index to test both the substring column and another column.

## **Creating Data Distributions**

You can use the **MEDIUM** or **HIGH** keywords with the **UPDATE STATISTICS** statement to specify the mode for data distributions on specific columns. These keywords indicate that Universal Server is to generate statistics about the distribution of data values for each specified column and place that information in a system catalog table called **sysdistrib**. If a distribution has been generated for a column, the optimizer uses that information to estimate the number of rows that match a query against a column. Data distributions in **sysdistrib** supersede values in the **colmin** and **colmax** column of the **syscolumns** system catalog table when estimating the number of rows returned.

When you use data-distribution statistics for the first time, try to update statistics in medium mode for all your tables, and then update statistics in high mode for all columns that head indexes. This strategy produces single-table query estimates that, on the average, have a margin of error less than 2.5 percent of the total number of rows in the table. (For columns with high-mode distributions, the default resolution is 1 percent.)

Unless column values change considerably, you need not regenerate the data distributions. To verify the accuracy of the distribution, compare **dbschema-hd** output with appropriately constructed **SELECT** statements.

For each table that your query accesses, build data distributions according to the following guidelines:

1. Run the UPDATE STATISTICS statement with the MEDIUM keyword and DISTRIBUTIONS ONLY clause for each table, or for the entire database. The default parameters are sufficient unless the table is large, in which case you should use resolution parameters of 1.0 and 0.99. This approach constructs distributions only. With the DISTRIBUTIONS ONLY clause in effect, the performance effect while you execute the statement is slightly more than that when you build distributions on a table-by-table basis. However, it requires less programming.
2. Run the UPDATE STATISTICS statement with the HIGH keyword for all columns that head an index. For the fastest execution time of the update statistics operation, issue a separate UPDATE STATISTICS HIGH statement for each column that heads an index. This approach also constructs index statistics about each column along with data distributions.
3. For each multicolumn index, issue an UPDATE STATISTICS LOW statement for all columns in the multicolumn index. This approach also constructs index statistics.

For additional information about data distributions and the UPDATE STATISTICS statement, see the *Informix Guide to SQL: Syntax*.

## Improving Query Performance with Indexes

You can often add or, in some cases, remove indexes to improve the performance of a query. For information on the structure of indexes and space requirements for indexes, refer to “Estimating Index Pages” on page 3-13.

Consider adding indexes in the following cases to improve the performance of a query:

- When the column is frequently used in filter expressions  
For more information, refer to “Filtered Columns” on page 4-35.
- When a function is frequently used in filter expressions  
For more information, refer to “Functions in Filters” on page 4-36.

- When the column is indexed and the value to be compared is a column in another table (a join expression)  
For more information, refer to “Join Expressions” on page 4-37.
- When the column is frequently used to order or group  
For more information, refer to “Order-By and Group-By Columns” on page 4-37.
- When the column does not involve duplicate keys  
For more information, refer to “Avoiding Columns with Duplicate Keys” on page 4-37.
- When the column is amenable to clustered indexing  
For more information, refer to “Clustering” on page 4-39.
- When the database server automatically builds an index  
For more information, refer to “Replacing Autoindexes with Permanent Indexes” on page 4-39.
- When multiple column or functional values are frequently used to order or group  
For more information, refer to “Using Composite Indexes” on page 4-39.

In addition, if you have one or more indexes on a table that has been updated extensively, you might want to drop and re-create an index. For more information, refer to “Dropping and Rebuilding Indexes After Updates” on page 4-41.

You also need to choose an index type that is best suited to the type of data that you need to index. For more information, refer to “Choosing an Index Type” on page 4-47.

### ***Filtered Columns***

For a query on a single table, the optimizer uses an index when the column is indexed and a value to be compared is a literal, a host variable, or an uncorrelated subquery. For this purpose, an indexed column is any single column with an index or the first column named in a composite index. Universal Server first finds the row in an appropriate index to locate relevant rows in the table. Without the index, Universal Server must scan each table in its entirety. For more information on composite indexes, refer to “Using Composite Indexes” on page 4-39.

The optimizer determines whether an index can be used to evaluate a filter. If the values listed in the index are all that is required, the rows are not read. It is faster to omit the page lookups for data pages whenever values can be read directly from the index.

If a column is often used to filter the rows of a large table, consider placing an index on it. The optimizer can use the index to select the desired columns and avoid a sequential scan of the entire table. One example is a table that contains a large mailing list. If you find that a postal-code column is often used to filter a subset of rows, consider putting an index on that column.

This strategy yields a net savings of time only when the selectivity of the column is high; that is, when only a small fraction of rows holds any one indexed value. Nonsequential access through an index takes several more disk I/O operations than sequential access does, so if a filter expression on the column passes more than a fourth of the rows, the database server might as well read the table sequentially. As a rule, an index on a filter column saves time in the following cases:

- The column is used in filter expressions in many queries or in slow queries.
- The column contains at least 100 unique values.
- Most column values appear in fewer than 10 percent of the rows.

## ***Functions in Filters***

In Universal Server, you can create a *functional index* on the resulting values of a function on one or more columns. The function can be a built-in function or a user-defined function. When you create a functional index, Universal Server computes the return values of the function and stores them in the index. Universal Server can locate the return value of the function in an appropriate index without executing the function for each qualifying column.

For more information on indexing user-defined functions, refer to “Using a Functional Index” on page 4-54.

## ***Join Expressions***

The optimizer uses an index when the column is indexed and the value to be compared is a column in another table (a join expression). Universal Server can use the index to find matching values. The following join expression shows such an example:

```
WHERE customer.customer_num = orders.customer_num
```

If rows of **customer** are read first, values of **customer\_num** can be applied to an index on **orders.customer\_num**.

## ***Order-By and Group-By Columns***

When a large quantity of rows must be ordered or grouped, the database server must put the rows in order. One way the database server does this is to select all the rows into a temporary table and sort the table. However, if the columns to be ordered are indexed with a B-tree index, the optimizer sometimes reads the rows in sorted order through the index and thus avoids a final sort.

If all the columns in the ORDER BY clause appear in the same sequence as within a single B-tree index, Universal Server can use the index to read the rows in their ordered sequence. Because the keys in a B-tree index are in sorted sequence, the B-tree index really represents the result of sorting the table. When you place a B-tree index on the ordering column or columns, you can replace many sorts during queries with a single sort when the B-tree index is created.

If all the columns in the GROUP BY clause appear in one B-tree index, Universal Server can read groups with equal keys from the B-tree index without requiring additional processing after the rows have been retrieved from their tables.

### ***Avoiding Columns with Duplicate Keys***

When duplicate keys are permitted in an index, entries that match a given key value are grouped in lists. The database server uses these lists to locate rows that match a requested key value. When the selectivity of the index column is high, these lists are generally short. But when only a few unique values occur, the lists become quite long and, in fact, can cross multiple leaf pages.

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

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

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

## ***Clustering***

Clustering is a way to arrange the rows of a table so their physical order on disk closely corresponds to the sequence of entries in the index.

When you know that a table is ordered by a certain index, the I/O is more efficient when the cluster index is used to scan the table. You can also be sure that when the table is searched on that column, it is read effectively in sequential order instead of nonsequentially. These points are covered in “The Cost of Nonsequential Access” on page 4-19.

For more information on how to maintain the clustering order, refer to “Maintaining Indexes” on page 3-32.

## ***Replacing Autoindexes with Permanent Indexes***

The database server sometimes creates an *autoindex* on a large table when the optimizer determines that it is more cost effective to build the index dynamically than to scan the whole table repeatedly when you execute a nested-loop join. If the query plan includes an *autoindex* path to a large table, you might improve performance if you add an index on that column. You can reasonably let Universal Server build and discard an index if you perform the query occasionally. If you perform a query regularly, you can save time if you create a permanent index.

## ***Using Composite Indexes***

The optimizer can use a composite index (one that covers more than one column or function) in several ways.

### Composite Indexes on Columns

An index on the columns **a**, **b**, and **c** (in that order) can be used in the following ways:

- To locate a particular row

An index specifies the first columns with equality filters and subsequent columns with range expressions (<, <=, >, >=) to locate a row. The following examples of filters use the columns in a composite index:

```
where a=1
where a>=12 and a<15
where a=1 and b < 5
where a=1 and b = 17 and c >= 40
```

The following examples of filters cannot use that composite index:

```
where b=10
where c=221
where a>=12 and b=15
```

- To replace a table scan when all of the desired columns are contained within the index  
A scan that uses the index but does not reference the table is termed a *key-only search*.
- To join column **a**, columns **ab**, or columns **abc** to another table
- To implement ORDER BY or GROUP BY on columns **a**, **ab**, or **abc**, but not on **b**, **c**, **ac**, or **bc**

Execution is most efficient when you create a composite index with the columns in order from most to least distinct. In other words, the column that returns the highest count of distinct rows when queried with the DISTINCT keyword of the SELECT statement should come first in the composite index.

If your application performs several long queries, each of which contains ORDER BY or GROUP BY clauses, you can sometimes add indexes that produce these orderings and do not require a sort to improve performance. For example, the following query sorts each column in the ORDER BY clause in a different direction:

```
SELECT * FROM t1 ORDER BY a, b DESC;
```



To avoid using temporary tables to sort column **a** in ascending order and column **b** in descending order, create a composite index on (**a**, **b** DESC) or on (**a** DESC, **b**). Create only one of these indexes because of the bidirectional traversal capability of the database server. For more information on the bidirectional traversal capability, refer to the *Informix Guide to SQL: Syntax*.

On the other hand, perform a table scan and sort the results instead of using the composite index when the following criteria are met:

- Your table is well ordered relative to your index.
- The number of rows retrieved by the query represents a large percentage of the available data.

### *Composite Indexes on Functions*

A functional index is an index that is defined on the value(s) that a user-defined function returns. For more information on functional indexes, refer to “Using a Functional Index” on page 4-54.

If you have a table with columns **a**, **b**, **c**, **d**, **e**, and **f**, you can define indexes on the following combinations of functions and columns:

- An index on a function defined on multiple columns, as in the following example:
- A composite index that consists of multiple functions, as in the following example:
- A composite index that consists of multiple functions and columns, as in the following examples:

```
create index f1 on tabl (function1(a,b)) ...
```

```
create index f2 on tabl (function1(a,b),
function2(c,d))...
```

```
create index f3 on tabl (a, function3(b,c)) ...
create index f4 on tabl
(a,function3(b,c),d,function4(e,f))...
```

### ***Dropping and Rebuilding Indexes After Updates***

When an update transaction commits, the Universal Server **btree** cleaner thread removes deleted index entries and, if necessary, rebalances the index nodes. However, depending on your application (in particular, the order in which it adds and deletes keys from the index), the structure of an index can become inefficient.

Use the **oncheck -pT** command to determine the amount of free space in each index page. If your table has relatively low update activity and a large amount of free space exists, you might want to drop and re-create the index with a larger value for **FILLFACTOR** to make the unused disk space available.

For more information on how Universal Server maintains an index tree, refer to the *INFORMIX-Universal Server Administrator's Guide*.

## **Reducing the Effect of Join and Sort Operations**

After you understand what the query is doing, look for ways to obtain the same output with less effort. The following suggestions can help you rewrite your query more efficiently:

- Avoid or simplify sort operations.
- Use parallel sorts.
- Use temporary tables to reduce sorting scope.

### ***Avoiding or Simplifying Sort Operations***

Sorting is not necessarily a liability. The sort algorithm is highly tuned and extremely efficient. It is as fast as any external sort program that you might apply to the same data. You need not avoid infrequent sorts or sorts of relatively small numbers of output rows.

Avoid or reduce the scope of repeated sorts of large tables. The optimizer avoids a sort step whenever it can produce the output in its proper order automatically with an index. The following factors prevent the optimizer from using an index:

- One or more of the ordered columns or returned value of functions is not included in the index.
- The columns are named in a different sequence in the index and the ORDER BY or GROUP BY clause.
- The ordered columns are taken from different tables.

Another way to avoid sorts is discussed in “Using Temporary Tables to Reduce Sorting Scope” on page 4-43.

If a sort is necessary, look for ways to simplify it. As discussed in “Time Used for a Sort” on page 4-16, the sort is quicker if you can sort on fewer or narrower columns.

### ***Using Parallel Sorts***

When you cannot avoid sorting, Universal Server takes advantage of multiple CPU resources to perform the required sort-and-merge operations in parallel. Universal Server can use parallel sorts for any query; parallel sorts are not limited to PDQ queries. The **PSORT\_NPROCS** environment variable specifies the maximum number of threads that can be used to sort a query.

When PDQ priority is greater than 0 and **PSORT\_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable to request a specific proportion of PDQ resources for a query. You can use the **MAX\_PDQPRIORITY** parameter to limit the number of such user requests. For more information on **MAX\_PDQPRIORITY**, refer to the *INFORMIX-Universal Server Administrator's Guide*.

In some cases, the amount of data that is sorted can overflow the memory resources allocated to the query and result in I/O to a dbspace or sort file. For more information, refer to the *INFORMIX-Universal Server Administrator's Guide*.

### ***Using Temporary Tables to Reduce Sorting Scope***

Building a temporary, ordered subset of a table can speed up a query. It can help to avoid multiple-sort operations and can simplify the work of the optimizer in other ways.

For example, suppose your application produces a series of reports on customers who have outstanding balances, one report for each major postal area, ordered by customer name. In other words, a series of queries occurs, each of the following form (with hypothetical table and column names):

```
SELECT cust.name, rcvbles.balance, ...other columns...
FROM cust, rcvbles
WHERE cust.customer_id = rcvbles.customer_id
      AND rcvbles.balance > 0
      AND cust.postcode LIKE '98_ _ _'
ORDER BY cust.name
```

This query reads the entire **cust** table. For every row with the correct postal code, Universal Server searches the index on **rcvbles.customer\_id** and performs a nonsequential disk access for every match. The rows are written to a temporary file and sorted. For more information on temporary files, refer to the *INFORMIX-Universal Server Administrator's Guide*.

This procedure is acceptable if the query is performed only once, but this example includes a series of queries, each of which incurs the same amount of work.

An alternative is to select all customers with outstanding balances into a temporary table, ordered by customer name, as the following example shows:

```
SELECT cust.name, rcvbles.balance, ...other columns...
FROM cust, rcvbles
WHERE cust.customer_id = rcvbles.customer_id
      AND rcvbles.balance > 0
INTO TEMP cust_with_balance
```

Now you can direct queries against the temporary table in this form, as the following example shows:

```
SELECT *
FROM cust_with_balance
WHERE postcode LIKE '98_ _ _'
ORDER BY cust.name
```

Each query reads the temporary table sequentially, but the table has fewer rows than the primary table.

## **Improving Sequential Scans**

You can improve performance of sequential read operations on large tables in the following ways:

- Eliminate repeated sequential scans.
- Use unions to avoid large scans.

### ***Eliminating Repeated Sequential Scans of Large Tables***

Sequential access to a table other than the first table in the plan is ominous because it threatens to read every row of the table once for every row selected from the preceding tables. You should be able to judge how many times that is: perhaps a few, but perhaps hundreds or even thousands.

If the table is small, reading it repeatedly is harmless because the table resides completely in memory. Sequential search of an in-memory table can be faster than searching the same table through an index, especially if those index pages are maintained in memory and push other useful pages out of the buffers.

When the table is larger than a few pages, however, repeated sequential access is deadly to performance. One way to prevent this problem is to provide an index to the column that is used to join the table.

Any user with the Resource privilege can build additional indexes. Use the CREATE INDEX statement to make an index.

An index consumes disk space proportional to the width of the key values and the number of rows. (See “Estimating Index Pages” on page 3-13.) Also, Universal Server must update the index whenever rows are inserted, deleted, or updated; this step slows these operations. If necessary, you can use the DROP INDEX statement to release the index after a series of queries, which frees space and makes table updates easier.

### ***Using Unions to Eliminate Unneeded Scans***

Certain forms of the WHERE clause force the optimizer to use sequential access, even when indexes exist on all the tested columns. The following query forces sequential access into the **orders** table:

```
SELECT * FROM orders
      WHERE (customer_num = 104 or customer_num = 1008)
      AND order_num >1732
```

The key element is that two (or more) separate sets of rows are retrieved. The sets are defined by relational expressions that are connected by AND. In the following example, one set is selected by this test:

```
(customer_num = 104 OR customer_num = 1008)
```

In the following example, the other set is selected by the test:

```
order_num > 1732
```

The optimizer uses a sequential access path even though indexes exist on the **customer\_num** and **order\_num** columns.

Queries of this form can be accelerated if you convert them into UNION queries. Write a separate SELECT statement for each set of rows and connect them with the UNION keyword. The following example shows a rewritten version of the preceding example:

```
SELECT * FROM orders
      WHERE (customer_num = 104 OR customer_num = 1008)

UNION ALL

SELECT * FROM orders
      WHERE order_num > 1732
```

The optimizer uses an index path for each query.

You can also rewrite the WHERE clause, as the following example shows:

```
WHERE (customer_num = 104 AND order_num > 1732) OR
      (order_num > 1732 AND customer_num = 1008)
```

## Reviewing the Optimization Level

You normally obtain optimum overall performance with the default optimization level, high. The time required to optimize the statement is usually unimportant. However, if experimentation with your application reveals that your query is still taking too long, you can set your optimization level to low and then check the SET EXPLAIN output to see if the optimizer chose the same query plan as before.

You can specify a high or low level of database server optimization with the SET OPTIMIZATION statement. This statement is described in detail in the *Informix Guide to SQL: Syntax*.

---

## Choosing an Index Type

In Universal Server, users can define their own data types and the functions that operate on these data types. DataBlade modules also provide new data types and functions to Universal Server users. In Universal Server, you can define indexes on the following kinds of user-defined data types:

- Opaque types

An *opaque data type* is a fundamental data type that you can use to define columns in the same way that you use built-in types. An opaque data type stores a single value and cannot be divided into components by the server. For information about opaque types, see the CREATE OPAQUE TYPE statement in the *Informix Guide to SQL: Syntax* and the documentation for your DataBlade module developer kit.

- Distinct types

A *distinct data type* has the same representation as, but is different from, an existing opaque or built-in data type. For information about distinct data types, see the *Informix Guide to SQL: Reference* and the description of the CREATE DISTINCT TYPE statement in the *Informix Guide to SQL: Syntax*.

For more information on the different data types, refer to the *Informix Guide to SQL: Reference*.

## Defining Indexes for User-Defined Data Types

As with built-in data types, you might improve the response time for a query when you define indexes for new data types. The response time for a query might improve when Universal Server uses an index for:

- columns used to join two tables.
- columns that are filters for a query.
- columns in an ORDER BY or GROUP BY clause.
- results of functions that are filters for a query.

For more information on when the query performance can improve with an index, refer to “Improving Query Performance with Indexes” on page 4-34.

Universal Server and DataBlade modules provide a variety of different types of indexes (also referred to as *secondary access methods*). A secondary access method is a set of server functions that build, access, and manipulate an index structure. These functions encapsulate index operations, such as how to scan, insert, delete, or update nodes in an index.

To create an index on a user-defined data type, you can use any of the following secondary access methods:

- **Generic B-tree index**  
A B-tree index is good for a query that retrieves a range of data values. For more information, see “The B-Tree Secondary Access Method” on page 4-48.
- **R-tree index**  
An R-tree index is good for searches on multidimensional data. For more information, see “Using a User-Defined Secondary Access Method” on page 4-52.
- **Secondary access method that a DataBlade module provides for a new data type**  
A DataBlade module that supports a certain type of data can also provide a new index for that new data type. For more information, see “Using an Index Provided by a DataBlade Module” on page 4-56.

You can create a functional index on the resulting values of a user-defined function on one or more columns. For more information, see “Using a Functional Index” on page 4-54.



Once you choose the desired index type, you might also need to extend an operator class for the secondary access method. For more information on how to extend operator classes, refer to the *Extending INFORMIX-Universal Server: Data Types* manual.

### ***The B-Tree Secondary Access Method***

Universal Server provides the *generic B-tree index* for columns in database tables. In traditional relational database systems, the B-tree access method handles only built-in data types and therefore can only compare two keys of built-in data types. The generic B-tree index is an extended version of a B-tree that Universal Server provides to support user-defined data types.

**Tip:** For more information on the structure of a B-tree index and how to estimate the size of a B-tree index, refer to “Estimating Index Pages” on page 3-13.

Universal Server uses the generic B-tree as the built-in secondary access method. This built-in secondary access method is registered in the **sysams** system catalog table with the name **btree**. When you use the CREATE INDEX statement (without the USING clause) to create an index, Universal Server creates a generic B-tree index. For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*.

**Tip:** Universal Server also defines another secondary access method, the *R-tree index*. For more information on how to use an R-tree index, see “Using a User-Defined Secondary Access Method” on page 4-52.

#### *Uses for a B-Tree Index*

A B-tree index is good for a query that retrieves a range of data values. If the data to be indexed has a logical sequence to which the concepts of *less than*, *greater than*, and *equal* apply, the generic B-tree index is a useful way to index your data. Initially, the generic B-tree index supports the relational operators (<, <=, =, >=, >) on all built-in data types and orders the data in lexicographical sequence.



The optimizer considers whether to use the B-tree index to execute a query if you define a generic B-tree index on:

- columns used to join two tables.
- columns that are filters for a query.
- columns in an ORDER BY or GROUP BY clause.
- results of functions that are filters for a query.

### *Extending a Generic B-Tree Index*

Initially, the generic B-tree can index data that is one of the built-in data types, and it orders the data in lexicographical sequence. However, you can extend a generic B-tree to support columns and functions on the following data types:

- *User-defined data types* (opaque and distinct data types) that you want the B-tree index to support

In this case, you need to extend the default operator class of the generic B-tree index.

- *Built-in data types* that you want to order in a different sequence from the lexicographical sequence that the generic B-tree index uses

In this case, you need to define a different operator class from the default generic B-tree index.

An *operator class* is the set of functions (operators) that are associated with a nontraditional B-tree index. For more details on operator classes, refer to “Choosing Operator Classes for Indexes” on page 4-57.

### *Determining the Available Access Methods*

Universal Server provides a built-in B-tree secondary access method. Your environment might have installed DataBlade modules that implement additional secondary access methods. If additional access methods exist, they are defined in the **sysams** system catalog table.

To determine the secondary access methods that are available for your database, query the **sysams** system catalog table with the following SELECT statement:

```
SELECT am_id, am_owner, am_name, am_type FROM sysams
WHERE am_type = 'S';
```

An 'S' value in the **am\_type** column identifies the access method as a secondary access method. This query returns the following information:

- The **am\_id** and **am\_name** columns identify the secondary access method.
- The **am\_owner** column identifies the owner of the access method.

#### ANSI

In an ANSI-compliant database, the access-method name must be unique within the name space of the user. The access-method name always begins with the owner in the format **am\_owner.am\_name**. ♦

By default, Universal Server provides the following definitions in the **sysams** system catalog table for two secondary access methods, **btree** and **rtree**.

Access Method	am_id Column	am_name Column	am_owner Column
Generic B-tree	1	btree	'informix'
R-tree	2	rtree	'informix'

**Important:** The **sysams** system catalog table does not contain a row for the built-in primary access method. This primary access method is internal to Universal Server and does not require a definition in **sysams**. However, the built-in primary access method is always available for use.

If you find additional rows in the **sysams** system catalog table (rows with **am\_id** values greater than 2), your database supports additional user-defined access methods. To determine whether a user-defined access method is a primary or secondary access method, check the value in the **am\_type** column.

For more information on the columns of the **sysams** system catalog table, see the *Informix Guide to SQL: Reference*. For information on how to determine the operator classes that are available in your database, see “Determining the Available Operator Classes” on page 4-61.



### ***Using a User-Defined Secondary Access Method***

The built-in secondary access method is a B-tree index. If the concepts of *less than*, *greater than*, and *equal* do *not* apply to the data to be indexed, then you probably want to consider a *user-defined secondary access method* that works with Universal Server. You can use a user-defined secondary access method to access other indexing structures such as an R-tree.

If your database supports a user-defined secondary access method, you can specify that the database server use this access method when it accesses a particular index. For information on how to determine the secondary access methods that your database defines, see “Determining the Available Access Methods” on page 4-50.

To choose a user-defined secondary access method, use the USING clause of the CREATE INDEX statement. The USING clause specifies the name of the secondary access method to use for the index that you create. This name must be listed in the **am\_name** column of the **sysams** system catalog table and must be a secondary access method (the **am\_type** column of **sysams** is 'S').

The secondary access method that you specify in the USING clause of CREATE INDEX must already be defined in the **sysams** system catalog. If the secondary access method has not yet been defined, the CREATE INDEX statement fails.

When you omit the USING clause from the CREATE INDEX statement, the database server uses B-tree indexes as the secondary access method. For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*.

#### ***The R-Tree Index***

Universal Server supports the *R-tree index* for columns that contain spatial data such as maps and diagrams. An R-tree index uses a tree structure whose nodes store pointers to lower-level nodes. At the leaves of the R-tree are a collection of data pages that store *n*-dimensional shapes. For more information on the structure of an R-tree index and how to estimate the size of an R-tree index, refer to “Estimating Index Pages” on page 3-13.

R-tree indexes are most beneficial when queries must find objects that are within other objects, objects that contain one or more objects, or objects that intersect other objects.

Universal Server automatically defines the R-tree access method in the **sysams** system catalog table with the name **rtree**.



**Important:** To use an R-tree index, you must install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements the R-tree index. These DataBlade modules implement the R-tree operator-class functions.

To create an R-tree index, you must include the USING clause of the CREATE INDEX statement. For example, if you have one of the spatial DataBlade modules installed, the following statement creates an R-tree index on the **location** column of the **buildings** table:

```
CREATE INDEX rtree_ix1 ON buildings (location) USING rtree;
```

For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*.

Universal Server databases define the R-tree secondary access method in the system catalog tables but do *not* provide the support functions to implement an R-tree index. If your system implements the **rtree** secondary access method, the CREATE INDEX statement in Figure 4-6 specifies the **rtree** secondary access method for the **clloc\_ix** index.

```
CREATE INDEX clloc_ix ON cust_tab(cust_loc)
IN dbspacel
USING rtree;
```

**Figure 4-6**  
*Specifying a User-Defined Secondary Access Method*

The CREATE INDEX statement in Figure 4-6 creates an index that uses the default operator class for the **rtree** secondary access method. You can also specify a new default operator class, if one exists, for an index. For more information, see “Using an Operator Class” on page 4-62.

### *Other User-Defined Secondary Access Methods*

A DataBlade module might define a secondary access method for a given type of data. When you install such a DataBlade, the installation process would define the secondary access method in the database, and you would see this access method in the **sysams** system catalog table. For information about any secondary access methods that a DataBlade module creates, see the relevant DataBlade guide.

### *Using a Functional Index*

Universal Server provides support for indexes on the following database objects:

- A column index  
You can create a column index on the actual values in one or more columns.
- A functional index  
You can create a functional index on the functional value of one or more columns.

To decide whether to use a column index or functional index, determine whether a column index is the right choice for the data that you want to index. An index on a column of some data types might not be useful for typical queries. For example, the following query asks how many images are dark:

```
SELECT COUNT(*) FROM photos WHERE darkness(picture) > 0.5
```

An index on the **picture** data itself does not improve the query performance. The concepts of *less than*, *greater than*, and *equal* are not particularly meaningful when applied to an image data type. Instead, a functional index that uses the **darkness()** function can improve performance. You might also have a user-defined function that executes frequently enough that performance improves when you create an index on its values.

### *What Is a Functional Index?*

When you create a functional index, Universal Server computes the values of the user-defined function and stores them as key values in the index. When a change in the table data causes a change in one of the values of an index key, Universal Server automatically updates the functional index.

You can use a functional index for functions that return values of both user-defined data types (opaque and distinct) and built-in data types. However, you cannot define a functional index if the function returns a large-object data type (BYTE or TEXT).

### *When Is a Functional Index Used?*

The optimizer considers whether to use a functional index to access the results of functions that are in one of the following query clauses:

- SELECT clause
- Filters in the WHERE clause

A functional index can be a B-tree index, an R-tree index, or a user-defined index type that a DataBlade module provides. For more information on the types of indexes, see “Defining Indexes for User-Defined Data Types” on page 4-47. For information on space requirements for functional indexes, refer to “Estimating Index Pages” on page 3-13.

### *How Do You Create a Functional Index?*

The function can be a built-in function or a user-defined function. If it is a user-defined function, it can be either an external function or a Stored Procedure Language function.

#### **To use a functional index**

1. To build a functional index on a user-defined function, you must first do the following:
  - a. Write the code for the user-defined function if it is an external function.
  - b. Register the user-defined function in the database with the CREATE FUNCTION statement.
2. Build the functional index with the CREATE INDEX statement.

The following steps create a functional index on the **darkness()** function:

1. Write the code for the user-defined **darkness()** function that operates on the data type and returns a decimal value.
2. Register the user-defined function in the database with the CREATE FUNCTION statement.

```
CREATE FUNCTION darkness(im image)
RETURNS decimal
EXTERNAL NAME '/lib/image.so'
LANGUAGE C NOT VARIANT
```

You can use the default operator class for the functional index because the return value of this **darkness()** function is a built-in data type, DECIMAL.

3. Build the functional index with the CREATE INDEX statement.

```
CREATE TABLE photos
(
    name char(20),
    picture image
    :
);
CREATE INDEX dark_ix ON photos (darkness(picture));
```

This example assumes that the user-defined data type of **image** has already been defined in the database.

The optimizer can now consider the functional index when you specify the **darkness()** function as a filter in the query:

```
SELECT count(*) FROM photos WHERE darkness(picture) > 0.5
```

You can also create a composite index with user-defined functions. For more information, see “Using Composite Indexes” on page 4-39.

### ***Using an Index Provided by a DataBlade Module***

In Universal Server, users can access new data types that are provided by DataBlade modules. A DataBlade module can also provide a new index for the new data type. For example, the Excalibur Text Search DataBlade module provides an index to search text data. For more information, refer to the *Excalibur Text Search DataBlade Module User's Guide*.



For more information on the types of data and functions that each DataBlade module provides, refer to the user guide for each DataBlade module. For information on how to determine the types of indexes available in your database, see “Determining the Available Access Methods” on page 4-50.

---

## Choosing Operator Classes for Indexes

For most situations, use the default operators that are defined for a secondary access method. However, when you want to order the data in a different sequence or provide index support for a user-defined data type, you must extend an operator class. For more information on how to extend an operator class, refer to the *Extending INFORMIX-Universal Server: Data Types* manual.

### What Is an Operator Class?

An *operator class* is a set of function names that is associated with a secondary access method. These functions allow the secondary access method to store and search for values of a particular data type. The query optimizer uses an operator class to determine if an index can process the query with the least cost. An operator class tells the Universal Server query optimizer two things:

- Which functions that appear in an SQL statement can be evaluated with a given index  
These functions are called the *strategy functions* for the operator class.
- Which functions the index uses to evaluate the strategy functions  
These functions are called the *support functions* for the operator class.

With the information that the operator class provides, the query optimizer can determine whether a given index is applicable to the query. The query optimizer can consider whether to use the index for the given query when the following conditions are true:

- An index exists on the particular column(s) in the query.
- For the index that exists, the operation on the column(s) in the query matches one of the strategy functions in the operator class associated with the index.

The query optimizer reviews the available indexes for the table(s) and matches the index keys with the column specified in the query filter. If the column in the filter matches an index key and the function in the filter is one of the strategy functions of the operator class, the optimizer includes the index when it determines which query plan has the least execution cost. In this manner, the optimizer can determine which index can process the query with the least cost.

Universal Server stores information about operator classes in the **sysopclasses** system catalog table.

### ***Strategy and Support Functions***

Universal Server uses the *strategy functions* of a secondary access method to help the query optimizer determine whether a specific index is applicable to a specific operation on a data type. If an index exists and the operator in the filter matches one of the strategy functions in the operator class, the optimizer considers whether to use the index for the query.

Universal Server uses the *support functions* of a secondary access method to build and access the index. These functions are not called directly by end users. When an operator in the query filter matches one of the strategy functions, the secondary access method uses the support functions to traverse the index and obtain the results. Identification of the actual support functions is left to the secondary access method.

### ***Default Operator Classes***

Each secondary access method has a *default operator class* that is associated with it. By default, the CREATE INDEX statement associates the default operator class with an index. For example, the following CREATE INDEX statement creates a B-tree index on the **zipcode** column and automatically associates the default B-tree operator class with this column:

```
CREATE INDEX zip_ix ON customer(zipcode)
```

For more information on how to specify a new default operator class for an index, see “Using an Operator Class” on page 4-62.

## Built-In B-Tree Operator Class

The built-in secondary access method, the generic B-tree, has a default operator class called **btree\_ops** that is defined in the **sysopclasses** system catalog table. By default, the CREATE INDEX statement associates the **btree\_ops** operator class with it when you create a B-tree index. For example, the following CREATE INDEX statement creates a generic B-tree index on the **order\_date** column of the **orders** table and associates with this index the default operator class for the B-tree secondary access method:

```
CREATE INDEX orddate_ix ON orders (order_date)
```

Universal Server uses the **btree\_ops** operator class to specify the following:

- The strategy functions to tell the optimizer which filters in a query can use a B-tree index
- The support function to build and search the B-tree index

### *The B-Tree Strategy Functions*

The **btree\_ops** operator class defines the following names of strategy functions for the **btree** access method:

- **lessthan** (<)
- **lessthanorequal** (<=)
- **equal** (=)
- **greaterthanorequal** (>=)
- **greaterthan** (>)

These strategy functions are all *operator functions*. That is, each function is associated with an operator symbol; in this case, with a relational-operator symbol. For more information on relational-operator functions, see the *Extending INFORMIX-Universal Server: Data Types* manual.

When the query optimizer examines a query that contains a column, it checks to see if this column has a B-tree index defined on it. If such an index exists *and* if the query contains one of the relational operators that the **btree\_ops** operator class supports, then the optimizer can choose a B-tree index to execute the query.

### ***The B-Tree Support Function***

The **btree\_ops** operator class has one support function, a comparison function called **compare()**. The **compare()** function is a user-defined function that returns an integer value to indicate whether its first argument is equal to, less than, or greater than its second argument, as follows:

- A value of 0 when the first argument is *equal to* the second argument
- A value less than 0 when the first argument is *less than* the second argument
- A value greater than 0 when the first argument is *greater than* the second argument

The B-tree secondary access method uses the **compare()** function to traverse the nodes of the generic B-tree index. To search for data values in a generic B-tree index, the secondary access method uses the **compare()** function to compare the key value in the query to the key value in an index node. The result of the comparison determines if the secondary access method needs to search the next-lower level of the index or if the key resides in the current node.

The generic B-tree access method also uses the **compare()** function to perform the following tasks for generic B-tree indexes:

- Sort the keys before the index is built
- Determine the linear order of keys in a generic B-tree index
- Evaluate the relational operators
- Search for data values in an index

Universal Server uses the **compare()** function to evaluate comparisons in the SELECT statement. To provide support for these comparisons for opaque data types, you must write the **compare()** function. For more information, see the *Extending INFORMIX-Universal Server: Data Types* manual.

Universal Server also uses the **compare()** function when it uses a B-tree index to process an ORDER BY clause in a SELECT statement. However, the optimizer does not use the index to perform an ORDER BY operation if the index does not use the **btree\_ops** operator class.

## Determining the Available Operator Classes

Universal Server provides the default operator class for the built-in secondary access method, the generic B-tree index. In addition, your environment might have installed DataBlade modules that implement other operator classes. All operator classes are defined in the **sysopclasses** system catalog.

To determine the operator classes that are available for your database, query the **sysopclasses** system catalog table with the following SELECT statement:

```
SELECT opclassid, opclassname, amid, am_name
FROM sysopclasses, sysams
WHERE sysopclasses.amid = sysams.am_id
```

This query returns the following information:

- The **opclassid** and **opclassname** columns identify the operator class.
- The **am\_id** and **am\_name** columns identify the associated secondary access methods.

By default, Universal Server provides the following definitions in the **sysopclasses** system catalog table for two operator classes, **btree\_ops** and **rtree\_ops**.

Access Method	opclassid Column	opclassname Column	amid Column	am_name Column
Generic B-tree	1	btree_ops	1	btree
R-tree	2	rtree_ops	2	rtree

If you find additional rows in the **sysopclasses** system catalog table (rows with **opclassid** values greater than 2), your database supports user-defined operator classes. Check the value in the **amid** column to determine to which secondary access methods the operator class belongs.

The **am\_defopclass** column in the **sysams** system catalog table stores the operator-class identifier for the default operator class of a secondary access method. To determine the default operator class for a given secondary access method, you can run the following query:

```
SELECT am_id, am_name, am_defopclass, opclass_name
FROM sysams, sysopclasses
WHERE sysams.am_defopclass = sysopclasses.opclassid
```

By default, Universal Server provides the following default operator classes.

Access Method	am_id Column	am_name Column	am_defopclass Column	opclass_name Column
Generic B-tree	1	btree	1	btree_ops
R-tree	2	rtree	2	rtree_ops

For more information on the columns of the **sysopclasses** and **sysams** system catalog tables, see the *Informix Guide to SQL: Reference*. For information on how to determine the access methods that are available in your database, see “Determining the Available Access Methods” on page 4-50.

## Using an Operator Class

The CREATE INDEX statement specifies the operator class to use for each component of an index. If you do not specify an operator class, CREATE INDEX uses the default operator class for the secondary access method that you create. You can use a *user-defined operator class* for components of an index. To specify a *user-defined operator class* for a particular component of an index, you can:

- use a user-defined operator class that your database already defines.
- use an R-tree operator class if your database defined the R-tree secondary access method.

### *Using an Existing Operator Class*

If your database supports multiple-operator classes for the secondary access method that you want to use, you can specify which operator classes the database server is to use for a particular index. For information on how to determine the operator classes that your database defines, see “Determining the Available Operator Classes” on page 4-61.

Each part of a composite index can specify a different operator class. You choose the operator classes when you create the index. In the CREATE INDEX statement, you specify the name of the operator class to use after each column or function name in the index-key specification. Each name must be listed in the **opclassname** column of the **sysopclasses** system catalog table and must be associated with the secondary access method that the index uses.

For example, if your database defines the **abs\_btree\_ops** secondary access method to define a new sort order, the following CREATE INDEX statement specifies that the **table1** table associates the **abs\_btree\_ops** operator class with the **col1\_ix** B-tree index:

```
CREATE INDEX col1_ix ON table1(col1 abs_btree_ops)
```

The operator class that you specify in the CREATE INDEX statement must already be defined in the **sysopclasses** system catalog table with the CREATE OPCLASS statement. If the operator class has not yet been defined, the CREATE INDEX statement fails. For information on how to create an operator class, see the next section.

### ***Using the R-Tree Index Operator Class***

Universal Server databases define an operator class called **rtree\_ops** for the R-tree secondary access method, **rtree**. This operator class is the default operator class for the **rtree** secondary access method. By default, the CREATE INDEX statement associates the **rtree\_ops** operator class with it when you create an R-tree index.

Universal Server uses the **rtree\_ops** operator class to specify the following:

- The strategy functions to tell the optimizer which filters in a query can use an R-tree index
- The support functions to build and search the R-tree index

Universal Server databases define the default R-tree operator class in the system catalog tables.



***Important:*** To use an R-tree index, install a spatial DataBlade module such as the Spatial DataBlade module, Geodetic DataBlade module, or any other third-party DataBlade modules that implement the R-tree index. These spatial DataBlade modules might provide their own operator class instead of **rtree\_ops**. For more information on the spatial DataBlade modules, consult the appropriate DataBlade module user guide.

If your system implements the **rtree** secondary access method, the CREATE INDEX statement in Figure 4-6 on page 4-53 creates an R-tree index on the **cust\_loc** column of the **cust\_tab** table and associates with it the default operator class for the R-tree secondary access method. If a second operator class, **rtree\_ops2**, has also been defined on the **rtree** secondary access method, the following CREATE INDEX statement specifies that the **col2\_rix** R-tree index associates the **rtree\_ops2** operator class with the **col2** column:

```
CREATE INDEX col2_rix ON table1(col2 rtree_ops)
USING rtree
```

For more information on the syntax of the CREATE INDEX statement, see the *Informix Guide to SQL: Syntax*.

---

## Managing PDQ Resource Requirements and Queries

This section discusses the parameters and strategies that you use to manage resources for PDQ. This section covers the following topics:

- How the optimizer structures a PDQ query
- The Memory Grant Manager
- Allocating Universal Server resources for PDQ
- Balancing resource requirements for OLTP and decision-support applications

### How the Optimizer Structures a PDQ Query

Depending on the number of tables or fragments that a query must search and the resources that are available for a decision-support query, the optimizer assigns different components of a query to different threads. These component threads, which are initiated by the **sqlexec** thread, are listed as *secondary threads* in the SET EXPLAIN output.



Secondary threads are further classified as either *producers*, or consumers, depending on their function. A producer thread supplies data to another thread. For example, a scan thread might read data from shared memory that corresponds to a given table and pass it along to a join thread. In this case, the scan thread is considered a producer, and the join thread is considered a consumer. The join thread, in turn, might pass data along to a sort thread. When it does so, the join thread is considered a producer, and the sort thread is considered a consumer.

Several producers can supply data to a single consumer. When this situation occurs, Universal Server sets up an internal mechanism, called an *exchange*, that synchronizes the transfer of data from those producers to the consumer. For instance, if a fragmented table is to be sorted, the optimizer typically calls for a separate scan thread for each fragment. Because of different I/O characteristics, the scan threads can be expected to complete at different times. An exchange is used to funnel the data produced by the various scan threads into one or more sort threads with a minimum amount of buffering. Depending on the complexity of the query, the optimizer can call for a multilayered hierarchy of producers, exchanges, and consumers. Consumer threads generally work in parallel with producer threads, so the amount of intermediate buffering that is performed by the exchanges remains negligible.

These threads and exchanges are created automatically and transparently. They terminate automatically as they complete processing for a given query. Universal Server creates new threads and exchanges as needed for subsequent queries.

## The Memory Grant Manager

The Memory Grant Manager (MGM) coordinates memory use, CPU virtual processors (VPs), disk I/O, and scan threads among decision-support queries. The MGM uses the DS\_MAX\_QUERIES, DS\_TOTAL\_MEMORY, DS\_MAX\_SCANS, and MAX\_PDQPRIORITY configuration parameters to determine the quantity of these PDQ resources that can be granted to a decision-support query. (For more information about these configuration parameters, refer to the *INFORMIX-Universal Server Performance Guide*.)

The MGM dynamically allocates the following resources for decision-support queries:

- The number of scan threads started for each decision-support query
- The number of threads that can be started for each query
- The amount of memory in the virtual portion of Universal Server shared memory that the query can reserve

When your Universal Server system has heavy OLTP use and you find performance is degrading, you can use the MGM facilities to limit the resources committed to decision-support queries. During off-peak hours, you can designate a larger proportion of the resources to parallel processing, which achieves higher throughput for decision-support queries.

Memory is granted to a query by the MGM for such activities as sorts, hash joins, sort-merge joins, and GROUP BY clauses. The amount of memory used by decision-support queries cannot exceed DS\_TOTAL\_MEMORY.

The MGM grants memory to queries in *quantum* increments. A quantum is calculated with the following formula:

$$\text{quantum} = \text{DS\_TOTAL\_MEMORY} / \text{DS\_MAX\_QUERIES}.$$

For example, if DS\_TOTAL\_MEMORY is 12 megabytes and DS\_MAX\_QUERIES is 4 megabytes, a quantum of memory equals 3 megabytes. In general, memory is allocated more efficiently when quanta are smaller. You can often improve performance of concurrent queries by increasing DS\_MAX\_QUERIES to reduce the size of a quantum of memory.

You can run the **onstat -g mgm** command to monitor resources allocated by the MGM. This command displays only the amount of memory that is currently used; it does not display the amount of memory that is granted. For more information about this command, refer to the *INFORMIX-Universal Server Administrator's Guide*.

The MGM also grants a maximum number of scan threads per query based on the values of the DS\_MAX\_SCANS and the DS\_MAX\_QUERIES parameters.

The following formula calculates the maximum number of scan threads per query:

$$\text{scan\_threads} = \min(nfrags, DS\_MAX\_SCANS * (pdqpriority / 100) * (MAX\_PDQPRIORITY / 100)).$$

*nfrags* is the number of fragments in the table that has the largest number of fragments.

*pdqpriority* is the PDQ priority value that is set by either the **PDQPRIORITY** environment variable or the SQL SET PDQPRIORITY statement.

For more information about any of these Universal Server configuration parameters, refer to the *INFORMIX-Universal Server Performance Guide* and the *INFORMIX-Universal Server Administrator's Guide*.

The **PDQPRIORITY** environment variable and the SQL SET PDQPRIORITY statement request a percentage of PDQ resources for a query.

You can use the MAX\_PDQPRIORITY configuration parameter to limit the percentage of the requested resources that a query can obtain and to limit the effect of decision-support queries on OLTP processing. For more information on MAX\_PDQPRIORITY, refer to “Limiting the Priority of DSS Queries” on page 4-68.

## Allocating Resources for PDQ Queries

When the Universal Server SQL query executor uses PDQ to perform a query in parallel, it puts a heavy load on the operating system. In particular, PDQ exploits the following resources:

- Memory
- CPU VPs
- Disk I/O (to fragmented tables, temporary table space)
- Scan threads

When you configure Universal Server, consider how the use of PDQ affects OLTP users, decision-support application users, and other application users.

You can control how Universal Server uses resources in the following ways:

- Limit the priority of PDQ queries.
- Adjust the amount of memory.
- Limit the number of scans threads.
- Limit the number of concurrent queries.

### ***Limiting the Priority of DSS Queries***

The default value for the PDQ priority of individual applications is 0, which means that PDQ processing is not used. Universal Server uses this value unless it is overridden by one of the following actions:

- The user sets the **PDQPRIORITY** environment variable.
- The application uses the SET PDQPRIORITY statement.

The **PDQPRIORITY** environment variable and the MAX\_PDQPRIORITY configuration parameter work together to control the amount of resources to allocate for parallel processing. Setting these configuration parameters correctly is critical for the effective operation of PDQ.

The MAX\_PDQPRIORITY configuration parameter allows the Universal Server administrator to set limits to the parallel processing resources consumed by DSS queries. Thus, the **PDQPRIORITY** environment variable sets a *reasonable* or *recommended* priority value, and MAX\_PDQPRIORITY limits the priority that an application can claim.

The MAX\_PDQPRIORITY configuration parameter specifies the maximum percentage of the requested resources that a query can obtain. For instance, if **PDQPRIORITY** is 80 and MAX\_PDQPRIORITY is 50, each active query receives an amount of memory equal to 40 percent of DS\_TOTAL\_MEMORY, rounded down to the nearest quantum. In this example, MAX\_PDQPRIORITY effectively limits the number of concurrent decision-support queries to two. Subsequent queries must wait for earlier queries to finish before they acquire the resources that they need to run.

An application or user can use the DEFAULT tag of the SET PDQPRIORITY statement to use the value for PDQ priority if the value has been set by the **PDQPRIORITY** environment variable. DEFAULT is the symbolic equivalent of a -1 value for PDQ priority.

You can use the **onmode** command-line utility to change the values of the following configuration parameters temporarily. These changes remain in effect only as long as Universal Server remains up and running. When Universal Server is initialized, it uses the values listed in the ONCONFIG file. Use **onmode** to make the following changes:

- Use **onmode -M** to change the value of DS\_TOTAL\_MEMORY.
- Use **onmode -Q** to change the value of DS\_MAX\_QUERIES.
- Use **onmode -D** to change the value of MAX\_PDQPRIORITY.
- Use **onmode -S** to change the value of DS\_MAX\_SCANS.

For more information about **onmode**, refer to the *INFORMIX-Universal Server Administrator's Guide*.

If you must change the values of the decision-support parameters on a regular basis (for example, to set MAX\_PDQPRIORITY to 100 each night to process reports), you can set the values with a scheduled operating-system job. For information about how to create scheduled jobs, refer to your operating-system manuals.

To obtain the best performance from Universal Server, choose values for the **PDQPRIORITY** environment variable and MAX\_PDQPRIORITY parameter, observe the behavior that results, and then adjust the values for these parameters. There are no well-defined rules for how to choose these environment variable and parameter values. The following sections discuss strategies for how to set **PDQPRIORITY** and MAX\_PDQPRIORITY for specific needs.

### *Limiting the Values of PDQPRIORITY*

The MAX\_PDQPRIORITY configuration parameter sets limits on the PDQ priority that Universal Server grants when users set the **PDQPRIORITY** environment variable or issue the SET PDQPRIORITY statement before they issue a query. When an application or an end user attempts to set a PDQ priority, the priority that is granted is multiplied by the value that is specified in MAX\_PDQPRIORITY.

Set the value of MAX\_PDQPRIORITY lower to allocate more resources to OLTP processing. Set the value higher to allocate more resources to decision-support processing. The possible range of values is 0 to 100. This range represents the percent of resources that you can allocate to decision-support processing.

### *Maximizing OLTP Throughput*

At times, you might want to allocate resources to maximize the throughput for individual OLTP queries rather than for decision-support queries. When this is the case, set **MAX\_PDQPRIORITY** to 0, which limits the value of PDQ priority to OFF. A PDQ priority value of OFF does not prevent decision-support queries from running. Rather, it causes the queries to run without parallelization. In this configuration, response times for decision-support queries might be slow.

### *Conserving Resources*

If applications make little use of queries that require parallel sorts and parallel joins, consider using the **LOW** setting for **PDQPRIORITY**.

If Universal Server operates in a multiuser environment, you might set **MAX\_PDQPRIORITY** to 1 to increase interquery performance at the cost of some intraquery parallelism. A trade-off exists between these two different types of parallelism because they compete for the same resources. As a compromise, you might set **MAX\_PDQPRIORITY** to some intermediate value (perhaps 20 or 30) and set **PDQPRIORITY** to **LOW**. This action sets the default behavior to **LOW** but allows individual applications to request more resources with the **SET PDQPRIORITY** statement.

### *Allowing Maximum Use of Parallelism*

Set **PDQPRIORITY** and **MAX\_PDQPRIORITY** to 100 if you want Universal Server to assign as many resources as possible to parallel processing. This setting is appropriate for times when parallel processing does not interfere with OLTP processing.

### *Determining the Level of Parallelism*

You can use different *number* settings of **PDQPRIORITY** to experiment with the effects of parallelism on a single application.

### *Limits on Parallelism Associated with PDQPRIORITY*

Universal Server reduces the PDQ priority of queries that contain outer joins to `LOW` (if set to a higher value) for the duration of the query. If a subquery or a view contains outer joins, Universal Server lowers only the PDQ priority of that subquery or view, not of the parent query or any other subquery.

Universal Server lowers the PDQ priority of queries that require access to a remote database (same or different Universal Server instance) to `LOW` if you set it to a higher value. In that case, all local scans are parallel, but all local joins and remote access are nonparallel.

### *Using Stored Procedures*

Universal Server *freezes* the PDQ priority that is used to optimize SQL statements within procedures at the time of procedure creation or the last manual recompilation with the `UPDATE STATISTICS` statement. You can embed the `SET PDQPRIORITY` statement within the body of your procedure to change the client value of **PDQPRIORITY**.

The PDQ priority value that Universal Server uses to optimize or reoptimize an SQL statement is the value that was set by a `SET PDQPRIORITY` statement, which must have been executed within the same procedure. If no such statement has been executed, the value that was in effect when the procedure was last compiled or created is used.

The PDQ priority value currently in effect outside a procedure is ignored within a procedure when it executes.

Informix suggests that you turn off PDQ priority when you enter a procedure and then turn it on again for specific statements. You can avoid tying up large amounts of memory for the procedure, and you can make sure that the crucial parts of the procedure use the appropriate PDQ priority, as the following example illustrates:

```
CREATE PROCEDURE my_proc (a INT, b INT, c INT)
    Returning INT, INT, INT;
SET PDQPRIORITY 0;
:
:
SET PDQPRIORITY 85;
SELECT ..... (big complicated SELECT statement)
SET PDQPRIORITY 0;
:
:
;
```

### ***Adjusting the Amount of Memory***

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

$$\text{DS\_TOTAL\_MEMORY} = p\_mem - os\_mem - rsdnt\_mem - (128K * users) - other\_mem.$$

<i>p_mem</i>	represents the total physical memory that is available on the host computer.
<i>os_mem</i>	represents the size of the operating system, including the buffer cache.
<i>rsdnt_mem</i>	represents the size of Informix-resident shared memory.
<i>other_mem</i>	is the size of memory used for other (non-Informix) applications.

The value for DS\_TOTAL\_MEMORY derived from this formula serves only as a starting point. To arrive at a value that makes sense for your configuration, you must monitor paging and swapping. (Use the tools provided with your operating system to monitor paging and swapping.) When paging increases, decrease the value of DS\_TOTAL\_MEMORY so that the OLTP workload can be processed.



The amount of memory that is granted to a single PDQ query is influenced by many system factors, but generally the amount of memory granted to a single PDQ query is proportional to the following formula:

$$\text{memory\_grant\_basis} = (\text{DS\_TOTAL\_MEMORY} / \text{DS\_MAX\_QUERIES}) * (\text{PDQPRIORITY} / 100) * (\text{MAX\_PDQPRIORITY} / 100)$$

### ***Limiting the Number of Concurrent Scans***

Universal Server apportions some number of scans to a query according to its PDQ priority (among other factors). DS\_MAX\_SCANS and MAX\_PDQPRIORITY allow you to limit the resources that users can assign to a query, according to the following formula:

$$\text{scan\_threads} = \min (nfrags, (\text{DS\_MAX\_SCANS} * (\text{pdqpriority} / 100) * (\text{MAX\_PDQPRIORITY} / 100) )).$$

*nfrags* is the number of fragments in the table with the largest number of fragments.

*pdqpriority* is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement.

For example, suppose a large table contains 100 fragments. With no limit on the number of concurrent scans allowed, Universal Server would concurrently execute 100 scan threads to read this table. In addition, as many users as wanted to could initiate this query.

As the Universal Server administrator, you set DS\_MAX\_SCANS to a value lower than the number of fragments in this table to prevent Universal Server from being flooded with scan threads by multiple decision-support queries. You can set DS\_MAX\_SCANS to 20 to ensure that Universal Server concurrently executes a maximum of 20 scan threads for parallel scans. Furthermore, if multiple users initiate PDQ queries, each query receives only a percentage of the 20 scan threads, according to the PDQ priority assigned to the query and the MAX\_PDQPRIORITY that the Universal Server administrator sets.

### ***Limiting the Maximum Number of Queries***

The DS\_MAX\_QUERIES configuration parameter limits the number of concurrent decision-support queries that can run. To estimate the number of decision-support queries that Universal Server can run concurrently, count each query that runs with **PDQPRIORITY** set to 1 or greater as one full query.

Universal Server allocates less memory to queries that run with a lower priority, so you can assign lower-priority queries a PDQ priority value that is between 1 and 30, depending on the resource effect of the query. The total number of queries with PDQ priority values greater than 0 cannot exceed DS\_MAX\_QUERIES.

## **Managing Applications**

The Universal Server administrator, the writer of an application, and the users all have some control over the amount of resources that Universal Server allocates to process a query. The Universal Server administrator exerts control with configuration parameters. The application developer or the user can exert control through an environment variable or SQL statement.

### ***Using SET EXPLAIN***

The output of the SET EXPLAIN statement shows decisions made by the query optimizer. It shows whether parallel scans are used, the maximum number of threads that are required to answer the query, and the type of join that is used for the query. You can use SET EXPLAIN to study the query plans of an application. You can restructure a query or use OPTCOMPIND to change how the optimizer treats the query.

### ***Using OPTCOMPIND***

The **OPTCOMPIND** environment variable and the **OPTCOMPIND** configuration parameter indicate the preferred join method, thus assisting the optimizer in selecting the appropriate join method for parallel database queries.

Set the **OPTCOMPIND** configuration parameter to influence the optimizer in its choice of a join method. The value that you assign to this configuration parameter is referenced only when applications do not set the **OPTCOMPIND** environment variable.

Set **OPTCOMPIND** to 0 if you want Universal Server to select a join method exactly as in versions before OnLine Dynamic Server 7.0. This option ensures compatibility with previous versions of OnLine Dynamic Server.

When you set this parameter, remember that an application with an isolation mode of Repeatable Read can lock all records in a table when it performs a hash join or sort-merge join. For this reason, Informix recommends that you set **OPTCOMPIND** to 1.

If you want the optimizer to make the determination for you based on costs, regardless of the isolation-mode setting of applications, set **OPTCOMPIND** to 2.

For more information on **OPTCOMPIND** and the different join methods, refer to “Selecting Table-Access Paths” on page 4-10.

### ***Using SET PDQPRIORITY***

The **SET PDQPRIORITY** statement allows you to set PDQ priority dynamically within an application. The PDQ priority value can be any integer from -1 through 100.

The PDQ priority set with the **SET PDQPRIORITY** statement supersedes the **PDQPRIORITY** environment variable.

The **DEFAULT** tag for the **SET PDQPRIORITY** statement allows an application to revert to the value for PDQ priority that the environment variable sets, if any. For more information about the **SET PDQPRIORITY** statement, refer to the *Informix Guide to SQL: Syntax*.

### ***User Control of Resources***

To indicate the PDQ priority of a query, a user sets the **PDQPRIORITY** environment variable or executes the **SET PDQPRIORITY** statement before it issues a query. In effect, this action allows users to request a certain amount of parallel-processing resources for the query.

The resources that a user requests and the amount that Universal Server allocates for the query can differ. This difference occurs when the Universal Server administrator uses the **MAX\_PDQPRIORITY** configuration parameter to put a limit on user-requested resources, as the following section explains.

### ***Universal Server Administrator Control of Resources***

To manage the total amount of resources that Universal Server allocates to PDQ queries, the Universal Server administrator sets the environment variable and configuration parameters that are discussed in the following sections.

#### *Controlling Resources Allocated to PDQ*

First, you can set the **PDQPRIORITY** environment variable. The queries that do not set the **PDQPRIORITY** environment variable before they issue a query do not use PDQ. In addition, set the **MAX\_PDQPRIORITY** configuration parameter to place a limit on user-specified PDQ priority levels.

When you set the **PDQPRIORITY** environment variable and **MAX\_PDQPRIORITY** parameter, you exert control over the resources that Universal Server allocates between OLTP and DSS applications. For example, if OLTP processing is particularly heavy during a certain period of the day, you might want to set **MAX\_PDQPRIORITY** to 0. This configuration parameter sets a limit on the resources requested by users who use the **PDQPRIORITY** environment variable, so PDQ is turned off until you reset **MAX\_PDQPRIORITY** to a nonzero value.

### *Controlling Resources Allocated to Decision-Support Queries*

You control the resources that Universal Server allocates to decision-support queries by setting the DS\_TOTAL\_MEMORY, DS\_MAX\_SCANS, and DS\_MAX\_QUERIES configuration parameters. In addition to setting limits for decision-support memory and the number of decision-support queries that can run concurrently, Universal Server uses these parameters to determine the amount of memory to allocate to individual decision-support queries as they are submitted by users. To do this, Universal Server first calculates a unit of memory called a quantum by dividing DS\_TOTAL\_MEMORY by DS\_MAX\_QUERIES. When a user issues a query, Universal Server allocates a percent of the available quanta equal to the PDQ priority of the query.

Set the DS\_MAX\_SCANS configuration parameter to limit the number of concurrent decision-support scans that Universal Server allows.

If your applications depend on a global setting for PDQ priority, you can define **PDQPRIORITY** as a shared-environment variable in the **informix.rc** file. For more information on the **informix.rc** file, see the *Informix Guide to SQL: Reference*.



# Using ontape to Back Up and Restore Data

Backing Up and Restoring Universal Server Data . . . . .	5-4
What Is an Archive? . . . . .	5-4
What Is a Logical-Log Backup? . . . . .	5-5
What Is a Universal Server Restore?. . . . .	5-7
Physical and Logical Restores . . . . .	5-7
Setting the ontape Configuration Parameters . . . . .	5-9
Setting ontape Parameters with a Text Editor . . . . .	5-10
Creating a Level-0 Archive After You Change Parameters for ontape . . . . .	5-10
Setting the Tape-Device Parameters. . . . .	5-10
Specifying Separate Devices for Logical-Log Backups and Archiving . . . . .	5-11
Using NUL for a Tape Device . . . . .	5-12
Setting the Tape-Block-Size Parameters . . . . .	5-13
Setting the Tape-Size Parameters. . . . .	5-13
Checking ontape Configuration Parameters . . . . .	5-14
Using ontape . . . . .	5-14
Syntax . . . . .	5-14
Exit Codes . . . . .	5-15
Creating an Archive . . . . .	5-15
What Are Archive Levels?. . . . .	5-15
Scheduling Archives . . . . .	5-17
Precautions Before You Create an Archive . . . . .	5-21
Performing an Archive . . . . .	5-23
If the Logical-Log Files Fill During an Archive. . . . .	5-25
Monitoring Archive Status . . . . .	5-26
Details of an Archive . . . . .	5-27

Backing Up Logical-Log Files . . . . .	5-34
Before You Back Up the Logical-Log Files . . . . .	5-34
Starting an Automatic Logical-Log Backup . . . . .	5-36
Starting and Stopping a Continuous Logical-Log File Backup . . . . .	5-37
Details of a Logical-Log File Backup . . . . .	5-38
Restoring Universal Server Data . . . . .	5-43
Choosing the Type of Physical Restore . . . . .	5-43
Cold, Warm, or Mixed Restore—Choosing a Universal Server Mode . . . . .	5-44
Performing a Restore . . . . .	5-49
Steps to Restore the Whole System . . . . .	5-50
Steps to Restore Selected Dbspaces . . . . .	5-53
Changing Database Logging Status . . . . .	5-55



**T**his chapter explains the central concepts of data recovery for Universal Server using the **ontape** utility. The **ontape** utility enables you to perform the following operations:

- Archive Universal Server data
- Back up logical-log files
- Restore Universal Server data from archives and the corresponding backed-up logical-log files
- Change the logging status of a database

This chapter covers the following topics:

- Concepts involved in backing up and restoring Universal Server data
- Setting **ontape** configuration parameters
- Using **ontape**

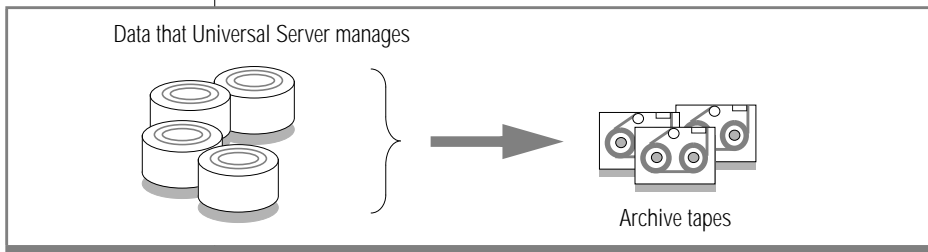
## Backing Up and Restoring Universal Server Data

The **ontape** utility enables you to back up Universal Server data and subsequently restore it if your current data becomes corrupt or inaccessible. The causes of corrupt or inaccessible data can range from a program error to a disk crash to a disaster that damages your entire facility. The **ontape** utility enables you to recover data lost due to such mishaps.

### What Is an Archive?

An *archive* is a copy of all or some portion of the data that Universal Server manages. More precisely, an archive is a copy of one or more Universal Server *dbspaces* (database *spaces*) or *blobspaces* and any supporting data that you might need to restore them. For a description of a Universal Server *dbspace*, see the *INFORMIX-Universal Server Administrator's Guide*.

The **ontape** utility enables you to create an archive of Universal Server data on tape or disk. To decrease the likelihood of both your current data and your archive being destroyed, you should store archives and other backup data in a safe location that is separate from your computer facility. Figure 5-1 illustrates the basic concept of a Universal Server archive.

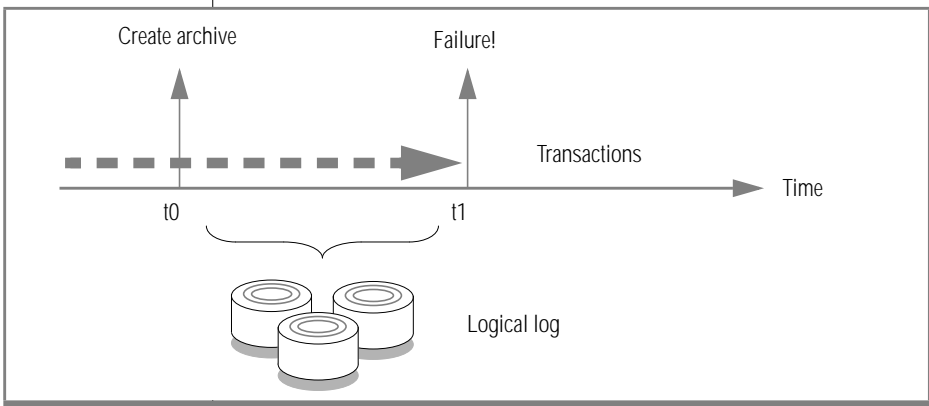


**Figure 5-1**  
*An Archive of  
Universal Server  
Data*

## What Is a Logical-Log Backup?

A *logical-log backup* is a copy to tape or disk of logical-log files that are full and eligible for backup. The logical-log files store a record of Universal Server activity that occurs *between archives*.

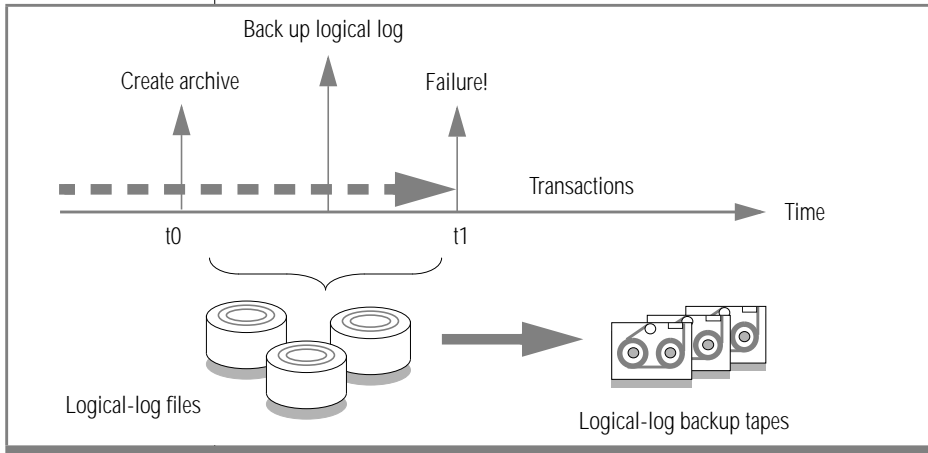
To illustrate, when you create an archive of all your Universal Server data at time **t0** and a failure occurs later at time **t1**, you could lose any transactions that occurred between those times. When you specify transaction logging for your databases, however, the transactions that occur between **t0** and **t1** are stored in the logical log. Figure 5-2 illustrates the function of the logical log.



**Figure 5-2**  
*The Logical Log*

## What Is a Logical-Log Backup?

Universal Server reuses the logical-log files to minimize the amount of disk space that is required for logging. Therefore, you must back up the logical-log files when they become full so that Universal Server can free them to make room for subsequent transactions. As with archives, you can back up the logical-log files to either tape or disk. For a complete description of the logical log, see the *INFORMIX-Universal Server Administrator's Guide*. Figure 5-3 illustrates a logical-log backup.



**Figure 5-3**  
*Logical-Log Backups*

When you restore dbspaces from your archives, in most cases you must also restore all transactions from the logical-log files that you backed up after the archive. When the failure causes Universal Server to come off-line, you can usually *salvage* any logical-log files that were not backed up at the time of the failure and then restore them as well. This action enables you to recover all of your Universal Server data up to the last complete transaction at the time of the failure.

## What Is a Universal Server Restore?

A Universal Server *restore* re-creates Universal Server data, particularly Universal Server dbspaces, from an archive and backed-up logical-log files.

### ***Physical and Logical Restores***

You must restore Universal Server data in two operations. The first operation is a *physical* restore, and the second is a *logical* restore. These two operations are defined as follows:

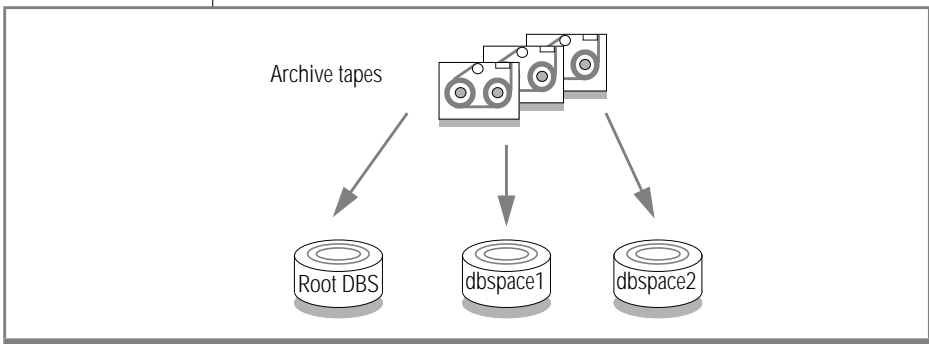
- **Physical restore**

A physical restore is the process of restoring dbspace or blobspace data from an archive.

- **Logical restore**

A logical restore accesses a logical-log backup to re-create in the restored dbspaces any transactions that were generated after the archive. When no databases use transaction logging, when you specified `\dev\null` as the backup device, or when you did not back up the logical-log files, you cannot restore any changes that you made to your databases after the archive.

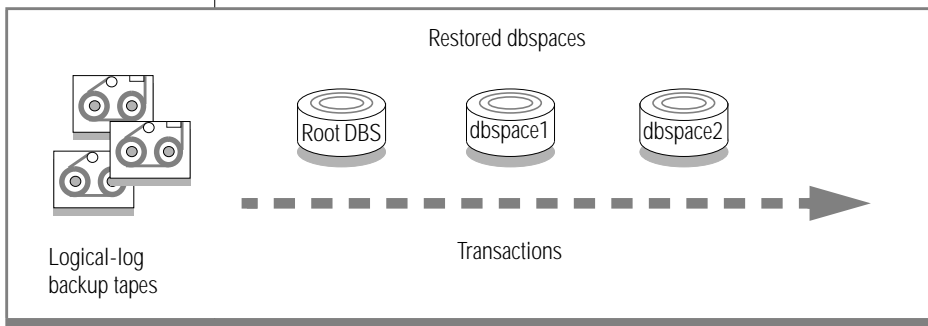
Figure 5-4 illustrates a physical restore.



**Figure 5-4**  
*Physical Restore*

You can perform a physical restore on either all Universal Server dbspaces and blobspaces or on selected dbspaces and blobspaces. For example, if you suffer a disk crash, you can restore to a new disk only those dbspaces with chunks that resided on the failed disk.

A *logical restore*, illustrated in Figure 5-5, restores Universal Server transactions from backed-up logical-log files for the dbspaces and blobspaces that were restored by the physical restore. Even when the dbspaces are physically restored from different archives, the logical restore rolls forward all the logical-log records for the dbspaces and blobspaces after the times of their respective archives.



**Figure 5-5**  
*Logical Restore*

If you do not specify transaction logging for your databases, if you specified NUL as the backup device, or if you have not backed up the logical-log files, any changes that you made to your databases after the archive cannot be restored.

The **ontape** utility performs both the physical and logical restore in a single operation.

## Setting the ontape Configuration Parameters

Six configuration parameters in the ONCONFIG file describe the devices that **ontape** uses to create archives and logical-log backups. The ONCONFIG file is located in %INFORMIXDIR%\etc and is specified by the ONCONFIG environment variable. For a description of the ONCONFIG environment variable and instructions on how to set it, see the *Informix Guide to SQL: Reference*. For descriptions of the **ontape** configuration parameters, see the *INFORMIX-Universal Server Administrator's Guide*.

The six ONCONFIG parameters that **ontape** uses are divided into two sets. The first set specifies the characteristics of the tape device and tapes for archives. The second set specifies the characteristics of the tape device and tapes for logical-log file backups.

The archive tape device and tape parameters are as follows:

TAPEDEV	represents the tape device used for archiving.
TAPEBLK	represents the block size of the tapes used for archiving, in kilobytes. The value must be greater than 8 kilobytes.
TAPESIZE	represents the size of the tapes used for archiving, in kilobytes.

The logical-log tape device and tape parameters are as follows:

LTAPEDEV	represents the logical-log tape device.
LTAPEBLK	represents the block size of tapes used for logical-log backups, in kilobytes.
LTAPESIZE	represents the size of tapes used for logical-log backups, in kilobytes.

To change the values of configuration parameters for **ontape**, you must be logged in as a member of the Informix-Admin group. You can use a text editor to set the values of parameters in the ONCONFIG file, or you can change the values of **ontape** parameters while Universal Server is in on-line mode. The change takes effect the next time that you start Universal Server. As the following sections explain, however, you have additional considerations when you change either the TAPEDEV parameter or the LTAPEDEV parameter to NUL.

## Setting ontape Parameters with a Text Editor

You can use a text-editor program to set the **ontape** parameters for either archiving or logical-log backup. Use the editor to open the ONCONFIG file, enter values for the parameters that you want to set, and save the file. The changes take effect the next time that you start Universal Server.

## Creating a Level-0 Archive After You Change Parameters for ontape

To ensure a proper restore, you must create a level-0 archive immediately after you change any of the archive or logical-log file backup tape device parameters unless you change the value to NUL. You must create the level-0 archive for the following two reasons:

- The Universal Server restore procedure with **ontape** cannot switch tape devices as it attempts to read the logical-log backup tapes. If the physical characteristics of the log-file tapes change during the restore, either because of a new block size or tape size, the restore fails.
- The restore fails if the tape device specified as TAPEDEV or LTAPEDEV at the time of the level-0 archive is unavailable when the restore begins.

The following sections contain information about how to set the tape-device, tape-block-size, and tape-size parameters for both archives and logical-log backups.

## Setting the Tape-Device Parameters

You must consider the following issues when you assign values to TAPEDEV and LTAPEDEV:

- Specifying separate devices for logical-log backups and archiving
- Using NUL for the tape device

The following sections explain each of these points.



### ***Specifying Separate Devices for Logical-Log Backups and Archiving***

If possible, the LTAPEDEV and TAPEDEV parameters in the ONCONFIG file should each specify a different device. When you specify separate devices for archives and logical-log backups, you can schedule archives and logical-log backups independently. You can create an archive on one device at the same time that you continuously back up the logical-log files on the other device.

If the LTAPEDEV and TAPEDEV parameters specify the same device, the logical log can fill and cause Universal Server to stop processing during an archive. When this situation occurs, your options are limited. You can either abort the archive to free the tape device and back up the logical-log files or leave normal processing suspended until the archive completes.

#### *Precautions to Take If You Use One Tape Device*

If you have only one tape device and you want to create archives while Universal Server is in on-line mode, you can take the following precautions:

- Configure Universal Server with a large amount of logical-log space through a combination of many log files or large log files. (For information about creating the logical-log files, see the *INFORMIX-Universal Server Administrator's Guide*.)
- Store all explicitly created temporary tables in a dedicated dbspace and then drop the dbspace before you archive.
- Create the archive when database activity is low.
- Free as many logical-log files as possible before you begin the archive.

The logical log still might fill before the archive completes. The archive is synchronized with a Universal Server checkpoint. An archive might have to wait for a checkpoint to synchronize activity, but the checkpoint cannot occur until all virtual processors exit from critical sections. If Universal Server processing is suspended because the logical-log file is full, the virtual processors cannot exit from their critical sections, and a deadlock results.

### ***Using NUL for a Tape Device***

If you specify NUL as an archive tape device, you can avoid the overhead of the level-0 archive that is required after some operations, such as changing the logging status of a database. Obviously, you cannot restore Universal Server data from an archive to NUL.

As described in “Do You Need to Back Up the Logical-Log Files?” on page 5-34, you can specify NUL as a tape device for logical-log backups if you decide that you do not need to recover transactions from the logical log.

If the tape device is specified as NUL, block size and tape size are ignored.

### ***Changing TAPEDEV to NUL***

The **ontape** utility reads the value of the TAPEDEV parameter at the start of processing. If you set TAPEDEV to NUL and request an archive, Universal Server bypasses the archive but still updates the dbspaces with the new archive time stamps. If you set TAPEDEV to NUL, you should do it before you start **ontape** to request the archive. You should not have a problem if you change TAPEDEV to NUL while Universal Server is in on-line mode but **ontape** is not running.

### ***Changing LTAPEDEV to NUL***

Take Universal Server off-line before you change the value of LTAPEDEV to NUL. If you make the change while Universal Server is in either quiescent or on-line mode, you can create a situation in which one or more log files are backed up but cannot be freed. This situation can interrupt processing because Universal Server stops if it finds that the next logical-log file (in sequence) is not free.

**Warning:** *If the ONCONFIG parameter LTAPEDEV is set to NUL, Universal Server marks the logical-log files as backed up as soon as they become full, effectively discarding logical log information.*



### *Verifying That the Tape Device Can Read the Block Size Specified*

Universal Server does not check the tape device when you specify the block size. Verify that the tape device specified by TAPEDEV and LTAPEDEV can read the block size that you specify for their block-size parameters. If not, you cannot restore the tape.

## Setting the Tape-Block-Size Parameters

Specify the block-size on the TAPEBLK and LTAPEBLK parameters as the largest block size, in kilobytes, permitted by your tape device.

If you specified the tape device as NUL, **ontape** ignores the block size.

Universal Server does not check that **ontape** can read the tape device at the time that you specify the block size. You should verify that Universal Server and **ontape** can read and write to the tape device with the block size that you specify. Otherwise, you might experience problems when you need to perform an archive, backup, or restore.

## Setting the Tape-Size Parameters

Set the tape size on the TAPESIZE and LTAPESIZE parameters using the number of blocks. These parameters specify the maximum number of blocks that the tape can hold.

If you specified the tape device as NUL, **ontape** ignores the tape size.

If you are doing continuous logical-log backup, the amount of data written to the tape is the smaller of LTAPESIZE and the following formula:

$$(\text{sum of space occupied by all logical-log files on disk}) - (\text{largest logical-log file})$$

This formula ensures that the I/O to the device completes and the logical-log files are freed before a log-full condition occurs.

## Checking ontape Configuration Parameters

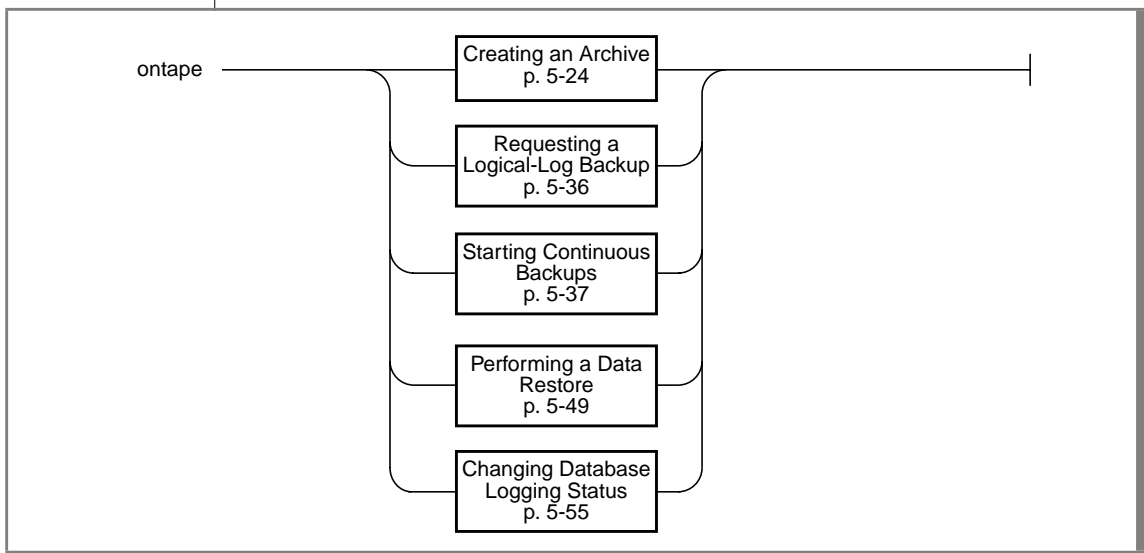
To examine the current values of the parameters that **ontape** uses in the ONCONFIG file, execute **onstat -c** while Universal Server is running.

## Using ontape

The **ontape** utility provides options that enable you to change the logging status of a database, create an archive, back up the logical-log files, and restore Universal Server data from archives and logical-log backups.

## Syntax

The following syntax diagram illustrates the basic operations of the **ontape** utility.



If **ontape** needs more than one tape during an archive or logical-log backup, it prompts you to mount each additional tape.

## ***Exit Codes***

The **ontape** utility uses the following two exit codes:

- 0 indicates a normal exit from **ontape**.
- 1 indicates an exception condition.

## **Creating an Archive**

This section explains how to plan for and create archives of your Universal Server data. It discusses the following topics:

- Archive levels
- Scheduling archives
- Precautions before you create an archive
- Performing an archive
- Monitoring archive status
- Details of an archive

### ***What Are Archive Levels?***

If Universal Server manages a large tape of data, it does not always make sense to archive all the data each time that you create an archive. For example, if some of your information changes frequently, but some remains stable, it seems inefficient to archive the stable information every time that you archive the volatile information.

Universal Server supports the following three *archive levels*:

- Level-0 archives all used pages.
- Level-1 archives all changes since the last level-0 archive.
- Level-2 archives all changes since the last level-1 archive.

The following sections explain the archive levels.

### *Level-0 Archives*

A level-0 archive is the baseline archive. It contains a copy of every used disk page (dbspace and blob space) that would be needed to restore the Universal Server database server to its state at that time. If a computer is completely destroyed (by fire or flood, for example), a level-0 archive is needed to completely restore Universal Server data on the replacement computer.

For on-line archives, the archive data reflects the contents of the dbspaces and blob spaces at the time that the level-0 archive began. The time that marks the beginning of the archive might be the same as the time of the last checkpoint. If no database activity has occurred since the last checkpoint, Universal Server uses the time of that checkpoint as the starting time for the archive. If database activity has occurred since the last checkpoint, Universal Server performs a new checkpoint and uses that as the starting time for the archive.

A level-0 archive can be time consuming.

### *Level-1 Archives*

A level-1 archive contains a copy of every changed page that contains data and system-overhead information since the last level-0 archive. All data copied to the archive reflects the state of the data at the time that the level-1 archive began. A level-1 archive usually takes less time than a level-0 archive because only part of the data that Universal Server manages is copied to the archive tape.

### *Level-2 Archives*

A level-2 archive contains a copy of every changed page that contains data and system-overhead information since the last level-1 archive. All data copied to the archive reflects the state of the data at the time that the level-2 archive began.

A level-2 archive after a level-1 archive usually takes less time than another level-1 archive because only the changes made since the last level-1 archive (instead of the last level-0) are copied to the archive tape.

## ***Scheduling Archives***

You should have a regular schedule for creating archives. Level-1 and level-2 archives are optional in your schedule, but level-0 archives are not. At the very least, the following administrative changes require a level-0 archive as part of the procedure. Consider waiting to make these changes until your next regularly scheduled level-0 archive:

- Changing TAPEDEV or LTAPEDEV from or to NUL requires an archive after the change is made.
- Adding logging to a database requires an archive after logging is added.
- Adding a dbspace or blobspace requires an archive. (The space cannot be restored by a level-1 or level-2 archive until after a level-0 archive).
- Starting mirroring for a dbspace that contains logical-log files requires an archive after the change to initiate mirroring.
- Adding a logical-log file requires an archive afterward to make the log file available.
- Dropping a logical-log file requires an archive after the log file is dropped.
- Moving one or more logical-log files to different dbspaces requires an archive after the logical-log file is dropped and after it is added.
- Changing the size or location of the physical log requires an archive after shared memory is reinitialized.
- Dropping a chunk requires an archive before you can reuse the dbspace that contains the chunk.

Figure 5-6 shows three different archive schedules, ranging from one that creates archives frequently to one that does not.

**Figure 5-6**  
*Examples of Archive Schedules*

Level	Daily Schedule	Weekly Schedule	Monthly Schedule
Level-0	Every night	Sundays	Once a month
Level-1	At lunch	Wednesdays	Once a week
Level-2	Every two hours	Mondays, Tuesdays, Thursdays, Fridays	Once a day

### *Determining Your Schedule Priorities*

Each of the following considerations affects the archive schedule that you create for your environment:

- Do you need to minimize the time for a restore?
- Do you need to minimize the time to create an archive?
- Do you need to create archives while Universal Server is in on-line mode?
- Do you need to use the same tape drive to create archives and back up logical-log files?
- Is the operator periodically unavailable?

### *Estimating the Time Required for an Archive*

You must consider several factors when you estimate the time that it takes to perform an archive. Each of the following items has an impact on the time that is needed to complete an archive:

- Overall speed of the tape device, including operating-system overhead
- Level of the archive
- Size of the archive
- Amount and type of database activity during the archive



- Amount and type of database activity in the period since the last archive
- Alertness of the operator to tape-changing demands

The best approach to estimating the time needed to complete an archive is to create an archive and use it as a gauge for the time that subsequent archives will require.

### *Minimizing the Size of an Archive*

The size of a level-1 archive is a function of the time and amount of update activity since your level-0 archive. The more often you create level-0 archives, and the less updating between archives, the smaller each level-1 archive is. Level-2 archives are also smaller if level-1 archives are more frequent and less updating occurs between them.

### *Minimizing the Time That an Archive Takes*

You can also reduce the duration of an archive by reducing the number of data pages that must be archived. You can reduce the number of data pages that must be archived by managing space for temporary tables in one of the following three ways:

- Create a temporary dbspace, as the *INFORMIX-Universal Server Administrator's Guide* describes, and store your temporary tables there. Any tables stored in a temporary dbspace are ignored during an archive.
- Create a normal dbspace in which you store all temporary tables; then drop the dbspace before you create an archive
- Drop all temporary tables before you archive the dbspaces in which they reside.

You need to perform a level-0 archive after you create a dbspace only if you plan to restore the dbspace. If you do not create a level-0 archive for the dbspace and a critical media failure occurs, one of the following two events happens:

- The dbspace is marked disabled, and the chunks go down.
- The dbspace is recovered when the logs are replayed because the statement that created the dbspace was logged.

If the dbspace is left in a disabled state, you can drop the dbspace and re-create it.

### *How to Minimize the Time for a Restore*

The time required to perform a restore is a function of the following factors:

- **Size and number of archives**

The minimum number of archives needed for a restore is one level-0 archive. The maximum number of archives is three, one of each archive level.

- **Amount of data to be restored**

- **Size and number of logical-log files since the last archive**

More log files take longer to restore.

- **Type of restore**

When you perform a full-system restore, you can restore some dbspaces first, while Universal Server is off-line. Then, when Universal Server comes on-line, those dbspaces are available while other dbspaces are being restored. This type of restore increases the availability of some dbspaces but also increases the total restore time. If your logical log becomes full, Universal Server suspends processing until you back it up. This fact means either that the archive must be aborted or processing remains suspended until the archive is complete and the logical-log files have been backed up.

You might consider the following strategy to minimize the time needed to restore the database server:

- Create a level-0 archive as often as is practical, perhaps every three days.
- Create a level-1 archive daily.
- Do not use level-2 archives.

The time that any possible restore requires is limited to the time needed to read and process the following data:

- A level-0 archive of the dbspace(s) being restored
- A level-1 archive, representing from one to three days of activity in the dbspaces being restored
- Logical-log files, representing less than a day's work in the dbspace or dbspaces being restored

### ***Precautions Before You Create an Archive***

Take the following precautions before you create an archive:

- Make sure you have sufficient logical-log space to create an archive.
- Keep a copy of your ONCONFIG file.
- Verify data consistency.
- Use the appropriate Universal Server mode.
- Ensure that an operator is available
- Label tapes appropriately.

The following sections address each of these topics.

#### *Ensure That You Have Enough Logical-Log Space*

If the total available space in the logical log (all the logical-log files) is less than half of a single log file, Universal Server does not create an archive. You must back up the logical-log files and attempt the archive again.

You cannot add a logical-log file or mirroring during an archive.

If only one tape device is available, make sure that all your logical-log files are backed up before you start your archive to reduce the likelihood of filling the logical log during the archive.

### *Keep a Copy of Your ONCONFIG File*

Keep a copy of the current ONCONFIG file when you create a level-0 archive. You need this information to restore Universal Server data from the archive tape.

### *Verify Data Consistency Before a Level-0 Archive*

To ensure the integrity of your archives, periodically verify that all Universal Server data and overhead information is consistent before you create a full-system level-0 archive. You need not check this information before every level-0 archive, but Informix recommends that you keep the necessary tapes from the most recent archive created immediately after Universal Server was verified to be consistent. For information on consistency checking, see the *INFORMIX-Universal Server Administrator's Guide*.

### *Use the Appropriate Universal Server Mode*

You can create an archive while Universal Server is in on-line or quiescent mode. The window from which you initiate the archive is dedicated to the archive (for displaying messages) until the archive is complete. Once you start an archive, Universal Server must remain in the same mode until the archive is finished; changing the mode terminates the archive activity.

An *on-line* archive is an archive that is created while Universal Server is in on-line mode. This type of archive is convenient if you want your Universal Server database server to be accessible while you are creating the archive.

Some minor inconveniences are associated with on-line archives. An on-line archive can slow checkpoint activity. This fact can contribute to a loss in performance. However, this decline in performance is far less costly than the time that would be lost if users were denied access to Universal Server during an archive.

During an on-line archive, allocation of some disk pages in dbspaces and blobspaces might be temporarily frozen. Disk-page allocation in dbspaces and blobspaces is blocked for one chunk at a time until the used pages in the chunk are archived.

A *quiescent* archive is created while Universal Server is in quiescent mode. Quiescent archives are useful when you want to eliminate partial transactions in an archive.

Quiescent archives might not be practical if users need continuous access to the databases that Universal Server manages.

### *Ensure That an Operator Is Available*

An operator should be available during an archive to mount tapes as prompted.

An archive might take several reels of tape. If an operator is not available to mount a new tape when one becomes full, the archive waits. During this time, if you perform the archive while Universal Server is in on-line mode, the physical log space might fill, causing Universal Server to abort the archive.

### *Label Tapes Appropriately*

When you label tapes created by **ontape**, the label should include the following information:

- Archive level
- Date and time
- Tape number provided by **ontape**

The following example shows how a label might look:

```
Level 1: Wed Apr 26, 1995 20:45 Tape # 3 of 5
```

Each archive begins with its first tape reel numbered 1, and each additional tape reel is numbered consecutively thereafter. A five-tape archive is numbered 1 through 5. (Of course, you might not know that it is a five-tape archive until it is finished.)

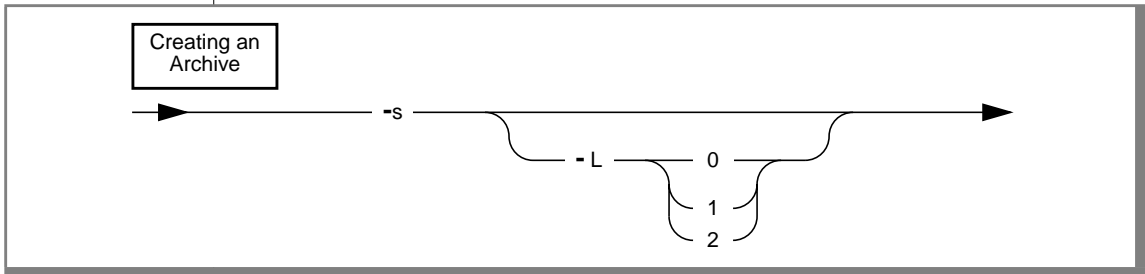
## ***Performing an Archive***

Take the following steps immediately before you begin an archive:

- Place a write-enabled tape on the tape-drive device that **TAPEDEV** specifies.
- Put the device on-line with the appropriate operating-system command.
- Place Universal Server in on-line or quiescent mode

Do not store more than one archive on the same tape; begin every archive with a new tape. (An archive is likely to span more than one tape.)

Use the **-s** option of the **ontape** command to create an archive.



**-s** directs **ontape** to create an archive.

**-L** directs **ontape** to create an archive of the level specified.

The **ontape** command prompts you to supply the archive level—0, 1, or 2—that you wish to create. If you use the **-L** option to specify the archive level as part of the command, you can avoid the prompt for the level.

An archive can require multiple tapes. After a tape fills, **ontape** rewinds the tape, displays the tape number for labeling, and prompts you to mount the next tape, if another one is needed. Follow the prompts for labeling and mounting new tapes. A message informs you when the archive is complete.

### Examples

Execute the following command to start an archive without specifying a level:

```
C:\> ontape -s
```

You can use the **-L** option to specify the level of the archive as part of the command, as the following example illustrates:

```
C:\> ontape -s -L 0
```

If you do not specify the archive level on the command line, **ontape** prompts you to enter it. Figure 5-7 illustrates a simple **ontape** archive session.

```
C:\> ontape -s
Please enter the level of archive to be performed (0, 1, or 2) 0

Please mount tape 1 on \\.\tape0 and press Return to continue ...
16:23:13 Checkpoint Completed: duration was 2 seconds
16:23:13 Level 0 Archive started on rootdbs
16:23:30 Archive on rootdbs Completed.
16:23:31 Checkpoint Completed: duration was 0 seconds

Please label this tape as number 1 in the arc tape sequence.
This tape contains the following logical logs:

3

Program over.
```

**Figure 5-7**  
*Example of a Simple  
Archive Created with  
ontape*

### ***If the Logical-Log Files Fill During an Archive***

If the logical log fills during an archive, Universal Server displays a message at the console and suspends normal processing. How you handle the filling of the logical log depends on whether you have one or two tape devices available.

If you have two tape devices available to Universal Server, log in as a member of the Informix-Admin group at an available Windows NT workstation.

Verify that LTAPEDEV and TAPEDEV specify different pathnames that correspond to separate tape devices. If they do, back up the logical-log files. See “Creating an Archive” on page 5-15.

If LTAPEDEV and TAPEDEV are identical, assign a different value to the logical-log tape device (LTAPEDEV) and initiate a logical-log file backup. This option is only a solution if the new value of LTAPEDEV is compatible with the block size and tape size used to create earlier logical-log file backups. (All tapes must reflect the physical characteristics specified at the time of the most recent level-0 archive.) Otherwise, your options are to either leave normal Universal Server processing suspended until the archive completes or cancel the archive.

If you are creating an archive with the only available tape device, you cannot back up any logical-log files until the archive is complete. If the logical-log files fill during the archive, normal Universal Server processing halts. You can either abort the archive to free the tape device and back up the logical logs to continue processing or leave normal processing suspended until the archive completes.

To prevent this situation, you can take the steps described in “Precautions to Take If You Use One Tape Device” on page 5-11.

### ***Monitoring Archive Status***

You can use **oncheck** to monitor the status of your recent archives.

If an archive is cancelled or interrupted, a slight chance exists that the archive progressed to the point where it is considered complete. If it is listed in the monitoring information, the archive completed.

To monitor your archive history with **oncheck**, execute **oncheck -pr** to display reserved-page information for the root dbspace. The last pair of reserved pages contains the following information for the most recent archive:

- Archive level (0, 1, or 2)
- Effective date and time of the archive
- Time stamp that describes when the archive began (expressed as a decimal)
- ID number of the logical log that was current when the archive began
- Physical location in the logical log of the checkpoint record that was written when the archive began

The effective date and time of the archive are the date and time of the checkpoint that the archive took as its starting point. This date and time might differ markedly from the time when the archive process was started.

For example, if no one had accessed Universal Server since Tuesday at 7 P.M., and you created an archive on Wednesday morning, the effective date and time for that archive would be Tuesday night, the time of the last checkpoint. In other words, if no activity has occurred since the last checkpoint, Universal Server does not perform another checkpoint at the start of the archive.



## ***Details of an Archive***

This section explains what Universal Server does during an archive. You do not need to understand this section to perform an archive; it is provided only as background information.

The following steps are involved in the process:

- The **ontape** utility connects and requests an archive.
- The **ontape** utility readies the device.
- Universal Server prepares to create an archive.
- Universal Server builds and sends archive data.
- The **ontape** utility writes data.
- The **ontape** utility and Universal Server commit the archive.

### *ontape Connects and Requests an Archive*

The **ontape** program sends a request to Universal Server to archive all dbspaces and blobspaces. The order in which the dbspaces are archived is as follows. Root dbspaces are archived first and, if any of the spaces to be archived are blobspaces, they are archived before any of the dbspaces. The blobpage allocation for each simple large object is blocked during the archive until the blobspace is archived. The block is released before the end of the rest of the archive. Universal Server recognizes when the blobspace is archived because it keeps track of what it is archiving.

Once **ontape** successfully connects with Universal Server and initiates the archive request, it reads archive backup data generated by Universal Server until no more archive data remains for the archive.

### *ontape Readies the Device*

The **ontape** program prompts you to mount a tape on the tape device specified in the configuration file.

### *Universal Server Prepares to Create an Archive*

Universal Server prepares to create an archive by performing the following tasks:

- If a blob space or db space is disabled, Universal Server returns an error and aborts the archive.
- Universal Server compares the specified archive level with the information in the archive reserved page.  
If Universal Server cannot find a record of a previous archive on the reserved page, the only valid archive level is a level-0 archive. Otherwise, any archive level is valid.
- Universal Server temporarily freezes the status of used logical-log files and checks the total amount of free log space. If free space is less than half of one log file, Universal Server refuses the archive request and recommends that you back up the logical-log files.
- Universal Server synchronizes with other archiving processes to guarantee that no two archiving processes are simultaneously archiving the same db space or blob space.
- Universal Server initiates a checkpoint (called the *archive checkpoint*). The checkpoint marks the beginning of the archive.

Universal Server uses a time stamp to determine which pages are to be archived. Universal Server does not archive a page that it creates after the archive checkpoint. If Universal Server modifies a page, Universal Server archives the before-image of the page from the physical log rather than the modified page.

For example, assume the checkpoint occurs at 3401. (Time stamps are not based on system time.) For a level-0 archive, all pages that contain time stamps less than 3401 must be archived. As Universal Server reads through disk pages during the archive, pages with time stamps greater than 3401 are ignored. Universal Server relies on the logical-log files to record modifications that occur after 3401.

The address of the most-recently written record in the current logical-log file is also noted during the checkpoint. This record becomes the last record from the logical-log file that is copied as part of this Universal Server archive.

Probably some transactions are ongoing during an on-line archive. The restore procedure describes how transactions that span the archive tape and the logical-log file are rolled back during a data restore, if necessary. For a description of how Universal Server restores data, see “Restoring Universal Server Data” on page 5-43.

- Universal Server reads archive history from the archive reserved page, saving the time stamp of the previous archive to set the criteria for determining which pages must be archived in non-level-0 archives.
- If the archive is a full-system archive, all the logical-log files that have log records from open transactions are marked so that they are not freed until they are archived.
- Universal Server builds a list of free pages within each chunk to be archived; the unused pages (and the pages devoted to the logical or physical log) at the time of the archive checkpoint are not archived. This information appears in entries on the dbspace chunk free-list pages and the blob space free-map page.
- Universal Server creates a temporary table for each dbspace being archived to store before-images from the physical log. These temporary tables are stored in the same spaces designated for other temporary tables. (For information on temporary tables, refer to the *INFORMIX-Universal Server Administrator's Guide*.) If Universal Server runs out of space in which to create the temporary tables, it aborts the archive.

Once Universal Server performs these tasks, it starts an internal archive thread that generates the archive data.

### *Universal Server Builds and Sends Data*

This section describes the order of information that is sent to **ontape**, and the special actions Universal Server takes during the archive to ensure that it archives completely and efficiently.

The first information Universal Server sends to **ontape** is a control page that contains information about the archive, including the following items:

- List of spaces included in the archive
- Archive level
- Archive time stamp
- Logging information

After the control page, Universal Server sends data in the following order:

1. A section that includes the reserve pages from the root dbspace is added to the archive.
2. A section including a snapshot of the logical logs that contain open transactions at the time of the archive checkpoint is added to the archive.
3. If the archive contains any blobspaces, they are included next. Archiving blobspaces early allows blobpage allocation to resume as soon as possible. The allocation of blobpages is blocked until the blobspace has been archived. Only the used portion of a blobpage is archived, not the whole page.
4. Following the blobspaces, the dbspaces are archived in no particular order.
5. The introductory tape-control page of each dbspace and blobspace section contains a mapping of the chunks contained within the space being backed up.
6. When a dbspace or blobspace is archived, the pages from the temporary table used to store before-images from the physical log are appended.
7. When Universal Server reaches the last page of the last chunk, the disk-reading portion of the archive procedure is complete. To mark the end of the archive data, Universal Server sends a trailer page to **ontape**.

Universal Server does not wait for **ontape** to consume archive data before it switches to threads that are doing other work. Universal Server continues with other processing during an archive.

When a page is updated while an archive is in progress, the archive process retrieves the before-image of the page from the physical-log file to capture the state of the page at the time of the archive checkpoint. Periodically, Universal Server empties the physical log of pages that are no longer needed for fast recovery. When this action occurs during an archive, any before-images that are needed for the archive are written to a temporary table. Once a before-image is written to the archive, Universal Server removes it from the temporary table. For a detailed description of physical logging, see the *INFORMIX-Universal Server Administrator's Guide*.

Because blobpages do not pass through shared memory, the strategy of archiving from the physical log is insufficient in itself. Universal Server must prevent clients from overwriting blobspace blobpages before they have been archived. To accomplish this task, Universal Server blocks allocation of blobpages in each blobspace chunk until all used blobpages in the chunk have been archived. As soon as the chunk is archived, blobpage allocation in that chunk resumes. This fact means that during an on-line archive, blobs cannot be inserted into a blobspace until the blobspace chunk has been archived.

Mirror chunks are not explicitly read for archiving. Pages within a mirror chunk are archived only if Universal Server cannot read the page from the primary chunk.

As Universal Server reads each disk page, it applies a set of criteria that determine which disk pages should be archived. Each page that meets the following criteria for archiving is copied to the archive tape:

- The page has been allocated.
- Universal Server uses the list of free pages in chunks that it created at the start of the archive to determine which pages were allocated.
- The page is not part of a logical-log file or the physical log.
- The page is needed for this archive level.

A level-0 archive requires Universal Server to archive all used disk pages that contain a time stamp less than the begin-archive checkpoint time stamp.

A level-1 archive directs Universal Server to archive all disk pages that contain a time stamp that is less than the archive checkpoint time stamp but greater than the time stamp associated with the most-recent level-0 archive.

A level-2 archive directs Universal Server to archive all disk pages that contain a time stamp that is less than the archive checkpoint time stamp but greater than the time stamp associated with the most-recent level-1 archive.

#### *ontape Writes Archive Data*

Before **ontape** receives archive data, it writes a tape header page to the archive device. The tape header page contains the following information:

- The tape-device block size (TAPEBLK)
- The size of tape (TAPESIZE)
- A flag that indicates that the tape is for an archive
- A time stamp that indicates the date and time of the archive
- The archive level
- The ID number of the logical-log file that contains the checkpoint record that began the archive
- The physical location of that checkpoint record in the logical-log file

#### *If the Archive Device is NUL*

If the archive device (TAPEDEV) is defined as NUL, **ontape** does not ask Universal Server to write a page to the device. Instead, **ontape** asks Universal Server to update the active PAGE\_ARCH reserved page with the same information that would have been written to the header page. For a description of the tape header page, see “ontape Writes Archive Data” on page 32. Universal Server also copies the checkpoint information to the active PAGE\_CKPT reserved page.

By means of this action, the root dbspace reserved pages receive acknowledgment that an archive has occurred. This event enables Universal Server to make use of newly added or changed resources.



**Warning:** A level-0 archive to NUL registers as a valid archive. Universal Server permits you to create a level-1 archive on a tape device even if your only level-0 archive was created when the archive device was NUL. Avoid this practice to avoid the risk of not having an archive from which you can restore your Universal Server data.

Setting the archive device to NUL can be a convenient practice, however, when you are setting up Universal Server by moving logs around, adding dbspaces, and loading tables. When Universal Server is ready for normal processing, however, you should set TAPEDEV to a proper device.

#### *ontape and Universal Server Commit the Archive*

When **ontape** has received all the archive backup data, it notifies Universal Server whether the archive should be committed or aborted. Committing a level-0 archive backup has the following implications:

- Newly mirrored dbspaces and blobspaces become available.
- If the archive includes the root dbspace, newly added log files become available.
- The archive becomes available for use during a restore.
- After a level-0 archive, if any database-logging changes are pending, the time of the database-logging change request is compared to the last level-0 archive for each dbspace and blobspace that makes up the database. If all the dbspaces and blobspaces were archived since the database-logging change request was made, the logging change takes effect, and database access is granted.

To commit the archive, Universal Server stores the history of the archive in the archive reserve pages. This information is used with subsequent archive increments.

When the archive is terminated, the temporary tables used for physical log pages are dropped.

## Backing Up Logical-Log Files

Use **ontape** to back up logical-log files only if you use **ontape** to make your archive tapes.

In addition to backing up logical-log files, you can use **ontape** to accomplish other tasks involved in maintaining and administering the logical log. For example, you might want to switch to the next log file, move logical-log files to other dbspaces, or change the size of the logical log. For instructions on these tasks, see the *INFORMIX-Universal Server Administrator's Guide*.

For a description of a logical-log backup, see “What Is a Logical-Log Backup?” on page 5-5.

For information on selecting a device for logical-log backups, see “Setting the ontape Configuration Parameters” on page 5-9.

### *Before You Back Up the Logical-Log Files*

Before you back up the logical-log files, you need to understand the following issues:

- Whether you need to back up the logical-log files
- When you need to back up the logical-log files

If you decide you need to back up the logical-log files, you must decide which type of backup you want to perform, automatic or continuous.

### *Do You Need to Back Up the Logical-Log Files?*

If you specify logging for your databases, Universal Server records transactions that occur between archives in the *logical log*, which consists of a finite number of logical-log files on disk. Universal Server continually needs to write new log records but also retain the log records that it has already written in case those transactions must be restored. To retain the records in the logical log, yet allow Universal Server to continue writing new log records in a finite amount of space, you must free full log files by copying them to a safe place on disk or tape.



Even if you do not use logging for any of your databases, you can still have log backups. These backups are small because they contain only administrative information such as checkpoint records and additions and deletions of chunks. By backing up these logical-log files, you can perform warm restores even if you are not using logging for any of your databases. For a description of a warm restore, see “A Warm Restore” on page 5-46.

You must keep the following two points in mind if you use simple large objects in a database that uses transaction logging:

- To ensure timely reuse of blobpages, you need to back up logical-log files. When users delete blobs in blobspaces, the blobpages are not freed for reuse until the log file that contains the delete records is freed. To free the log file, you must back it up.
- If a blobspace that needs to be backed up is unavailable during the backup, **ontape** skips it, so that it is impossible to recover the simple large object if that should become necessary. (However, blobpages from deleted blobs do become free when the blobspace is again available, even though the simple large object was not backed up.)

In addition, regardless of whether the database uses transaction logging, when you create a blobspace or add a chunk to a blobspace, the blobspace or new chunk is not available for use until the log file that records the event is not the current log file. For information on switching log files, see the *INFORMIX-Universal Server Administrator's Guide*.

If you decide that you do not need to recover transactions or administrative database activities between archives, you can set the Universal Server configuration parameter LTAPEDEV to NUL.



**Warning:** *If you set LTAPEDEV to NUL, this action has the following implications:*

- *You are only able to restore the data managed by your Universal Server database server up to the point of your most-recent archive and any previously backed-up logical-log files.*
- *When you recover, you must always perform a full-system restore. (See “A Full-System Restore” on page 5-43.) You cannot perform partial restores or restore when Universal Server is in on-line mode.*

When LTAPEDEV is set to NUL, Universal Server marks a logical-log file as backed up (status B) as soon as it becomes full. When the last open transaction in the log is closed, the log file is marked free (status F). Universal Server can then reuse that log file without waiting for it to be backed up. As a result, no logical-log records are preserved.

Other Universal Server mechanisms that use the logical log, such as fast recovery and rolling back transactions, are not impaired if you use NUL as your log-file backup device. For a description of Universal Server fast recovery, see the *INFORMIX-Universal Server Administrator's Guide*. For information about rolling back transactions, see the ROLLBACK WORK statement in the *Informix Guide to SQL: Syntax*.

### *When You Need to Back Up the Logical-Log Files*

You should attempt to back up each logical-log file as soon as it fills. A logical-log file that is ready to be backed up has a status of *used*. For information on monitoring the status of logical-log files, see the *INFORMIX-Universal Server Administrator's Guide*.

### ***Starting an Automatic Logical-Log Backup***

Universal Server can be in on-line mode when you back up logical-log files. To back up the logical-log files, use the **-a** option of the **ontape** command.

Requesting a  
Logical-Log Backup

—▶ — -a —▶

**-a** directs **ontape** to back up all full logical-log files.

The **-a** option backs up all full logical-log files and prompts you with an option to switch the log files and back up the formerly current log.

If the tape mounted on LTAPEDEV becomes full before the end of the logical-log file, **ontape** prompts you to mount a new tape.

If you press the Interrupt key while a backup is under way, Universal Server finishes the backup and then returns control to you. Any other full log files are left with a used status.

### Example

To back up all full logical-log files, execute the following command:

```
C:\> ontape -a
```

### ***Starting and Stopping a Continuous Logical-Log File Backup***

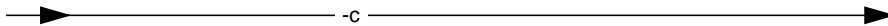
If you do not want to monitor the log files and start backups when the log files become full, you can start a continuous backup.

When you start a continuous backup, Universal Server automatically backs up each logical-log file as it becomes full. If you perform continuous logical-log file backups, you are protected against ever losing more than a partial log file, even in the worst-case media failure when a chunk that contains logical-log files fails.

If you use continuous backups, you do not have to remember to back up the log files, but someone must ensure that media is always available for the backup process. Also, the backup device and a window must be dedicated to the backup process.

To start a continuous backup of the logical-log files, use the **-c** option of the **ontape** command.

Starting Continuous Backups



**-c** directs **ontape** to initiate continuous backup of logical-log files.

The **-c** option initiates continuous logging. Universal Server backs up each logical-log file as it becomes full. Continuous backup does not back up the current log file.

Universal Server can be in on-line mode when you start continuous backups. Execute the following command to start continuous logging:

```
C:\> ontape -c
```

If the tape mounted on LTAPEDEV becomes full before the end of the logical-log file, **ontape** prompts you to mount a new tape.

### *Stopping a Continuous Logical-Log File Backup*

To end continuous logical-log file backup, press the Interrupt key (CTRL-C).

If you press the Interrupt key while Universal Server is backing up a logical-log file to a local device, all logs that were completely backed up before the interrupt are captured on the tape and are marked as backed up by Universal Server.

If you press the Interrupt key while Universal Server is waiting for a log file to fill (and thus is not backing up any logical-log files), all logs that were backed up before the interrupt are on the tape and marked as backed up by Universal Server.

After you stop continuous logging, you must start a new tape for subsequent log-backup operations.

You must explicitly request logical-log backups (using **ontape -a**) until you restart continuous logging.

### *Details of a Logical-Log File Backup*

This section describes the steps that **ontape** and Universal Server perform while backing up a logical-log file to tape. You need not understand this section to back up logical-log files; it is provided for information only.

The following list describes the steps:

1. The **ontape** utility connects and requests a log-file backup.
2. The **ontape** utility readies the device and tape.
3. Universal Server prepares to back up a logical-log file.

4. Universal Server builds and sends logical-log file data.
5. The **ontape** utility writes the logical-log backup data.
6. The **ontape** utility and Universal Server commit the backup.

The following sections explain each step in detail.

#### *ontape Connects and Requests a Backup*

The **ontape** program connects to Universal Server and sends a request to back up the logical-log files.

#### *ontape Readies the Device and Tape*

The **ontape** program prompts you to mount a tape on the tape device specified in the configuration file.

If the tape is new, **ontape** writes a tape header (also called a volume header) to the device.

#### *Universal Server Prepares to Back Up a Logical-Log File*

When Universal Server receives a request for a log-file backup, it locates the oldest logical-log file that has been used but not backed up (status U if you run **onstat -l**). Universal Server also checks to see that no other log backups are occurring.

Next Universal Server starts an internal thread that collects the log-file data and sends it to **ontape**.

*Universal Server Builds and Sends Logical-Log Data*

Universal Server builds the log-file data that needs to be backed up and sends it to **ontape** in the following sequence:

- **Blobpages**

Universal Server begins by comparing the identification number of the log file that it is backing up with every blobspace. (It actually looks at every blobspace free-map page.) Universal Server looks for blobpages that were allocated or marked for deletion during the time that this logical-log file was the current log file.

If there are blobpages to copy, each blobpage that was allocated or marked for deletion during the time that this log file was current is sent to the client. Universal Server precedes each blobpage with a header and follows it with a trailer.

If a blobspace containing blobpages that need to be backed up is unavailable at the time the logical-log backup occurs, Universal Server does not wait for the blobspace to become available.

Universal Server continues the backup without copying the blobpages that it needs. Thus, the simple large object cannot be restored when the logical-log file is rolled forward, and the simple large object is lost during a restore.

- **Log header**

After all blobs are checked and the required blobpages are sent to **ontape**, Universal Server creates a log header and sends it to **ontape**.

The log header is distinct from the tape header. The log header specifies (among other things) the ID number of the logical-log file and the number of pages from the logical-log file that need to be copied.

- Log records

Following the log header, Universal Server begins sending each page in the logical-log file that it is backing up. If some pages in the log file are not used (for example, if a file is backed up before it is full), the unused pages in the log file are not written to tape.

- Log trailer

After it sends the last page in the log file, Universal Server sends **ontape** a log trailer.

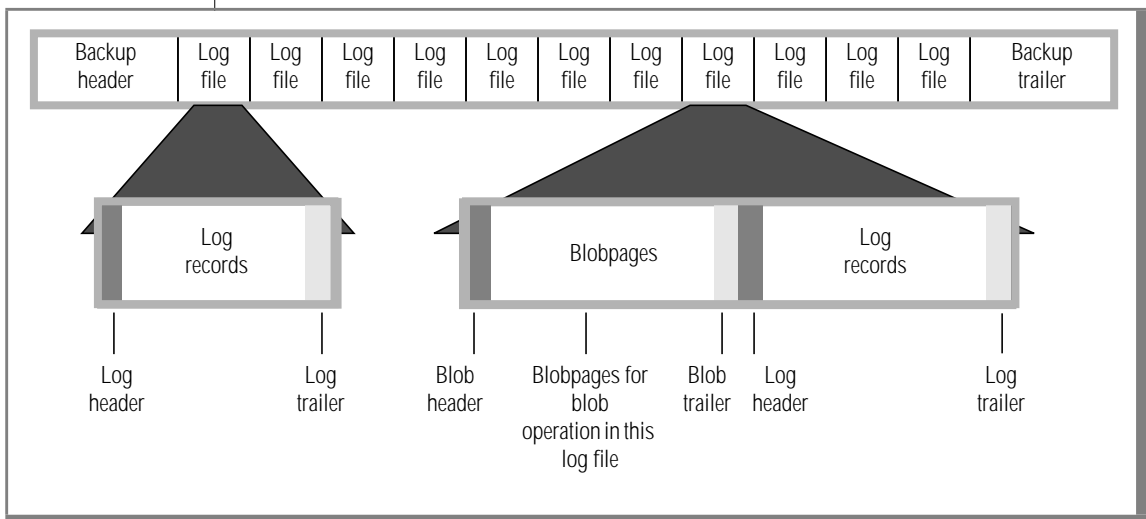
### *ontape Writes Data to the Backup Device*

The **ontape** program writes the logical-log data to the backup device in the same order that it receives it from Universal Server. This order is as follows:

- Blobpages
- Log-file header
- Log-file records
- Log-file trailer

Figure 5-8 illustrates the order of information on the logical-log backup tape.

**Figure 5-8**  
*Logical-Log Backup Tape Format*



### *ontape and Universal Server Commit the Backup*

When **ontape** has written all the log-file backup data, it notifies Universal Server whether the log-file backup should be committed or aborted. Committing the log-file backup changes the status to backed up. Aborting the log-file backup leaves the log file in the same state as it was prior to the log-file backup. When it changes the log-file status, Universal Server checks to see if the log file can be freed for reuse.

After a logical-log file is successfully backed up, **ontape** determines whether another log file needs to be backed up. If no other log files remain to be backed up and the user specified that the current log file should be backed up, Universal Server switches the current log file to the next log file and backs up the formerly current log file. If the log backup continues, it proceeds by repeating the steps outlined in the following three sections: “Universal Server Builds and Sends Logical-Log Data” on page 5-40, “ontape Writes Data to the Backup Device” on page 5-41, and “ontape and Universal Server Commit the Backup” on page 5-42.

If no more logs remain to be backed up and **ontape** is performing a continuous log backup, **ontape** goes into a loop in which it waits for a period of time and then again asks Universal Server if there are full log files to back up. As the log files fill, they are backed up.

When the entire log-backup process is complete, **ontape** writes a backup trailer to indicate the end of the backup session.

### *If a New Tape Is Needed*

If more than one tape is needed during the logical-log backup, **ontape** provides you with labeling information for the full tape and prompts you to mount a new tape.



## Restoring Universal Server Data

This section provides instructions for restoring Universal Server data with **ontape**. It provides instructions for the following procedures:

- A full-system restore
- A restore of selected dbspaces or blobspaces

Before you start restoring data, you should understand the concepts in “What Is a Universal Server Restore?” on page 5-7. As explained in that section, a complete recovery of Universal Server data generally consists of a physical restore and a logical restore.

### *Choosing the Type of Physical Restore*

If you are about to restore Universal Server data spaces because of a failure of Universal Server to go to *off-line mode*, you must restore all the data that Universal Server manages. This type of restore is called a *full-system* restore. If the failure did not cause Universal Server to go to off-line mode, you have the option of restoring only selected dbspaces and blobspaces; that is, only those that the failure affected.

#### *A Full-System Restore*

If Universal Server goes to off-line mode because of a disk failure or corrupted data, it means that a *critical dbspace* was damaged. The dbspaces in the following list are considered critical dbspaces:

- The root dbspace
- The dbspace that contains the physical log
- A dbspace that contains logical-log files

When you need to restore any critical dbspace, you must perform a full-system restore to restore all the data that your database server manages. You must start a full-system restore with a *cold restore*. See “Cold, Warm, or Mixed Restore—Choosing a Universal Server Mode” on page 5-44.

### *Restoring Selected Dbspaces and Blobspaces*

If your Universal Server database server *does not* go to off-line mode because of a disk failure or corrupted data, the damage occurred to a dbspace or blobspace that is not critical.

If you do not need to restore a critical dbspace, you can restore only those dbspaces and blobspaces that contain a damaged chunk or chunks. When a media failure occurs in one chunk of a dbspace or blobspace that spans multiple chunks, all active transactions for that dbspace or blobspace must terminate before Universal Server can restore it. You can start a restore operation before the transactions are finished, but the restore is delayed until Universal Server verifies that all transactions that were active at the time of the failure have finished.

### ***Cold, Warm, or Mixed Restore—Choosing a Universal Server Mode***

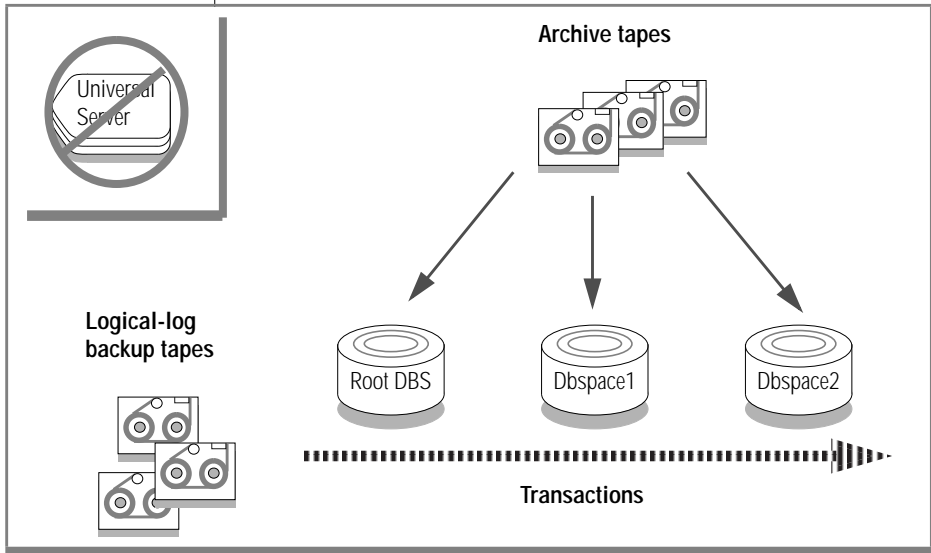
When you restore Universal Server data, you must decide whether you will do it while Universal Server is in off-line mode or on-line mode. This decision is not completely arbitrary, however. It depends in part on the data that you are restoring. The following sections explain the factors that determine which Universal Server mode you should use when you perform a restore.

#### *A Cold Restore*

You perform a *cold restore* while Universal Server is in off-line mode. It consists of both a physical restore and a logical restore. (For descriptions of a physical restore and a logical restore, see “What Is a Universal Server Restore?” on page 5-7.) You must perform a cold restore to restore any critical dbspaces.

As Figure 5-9 shows, you can restore all the dbspaces and blobspaces that Universal Server manages (a full-system restore) with a complete cold restore.

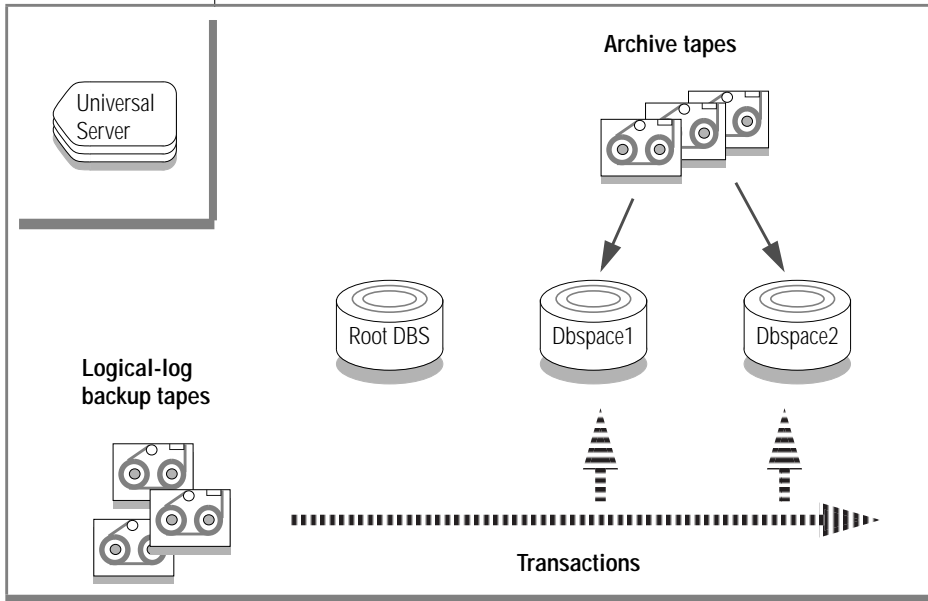
**Figure 5-9**  
*A Full-System Cold Restore*



Universal Server is off-line when you begin a cold restore, but it goes into recovery mode after the reserved pages have been restored. From that point on, it stays in recovery mode until either a logical restore is completed (after which step it is in quiescent mode) or you use the **onmode** utility to place it in another mode.

### A Warm Restore

A *warm restore* restores noncritical dbspaces and blobspaces while Universal Server is in on-line or quiescent mode. It consists of one or more physical restore operations (if you are restoring multiple dbspaces or blobspaces concurrently), a logical-log backup, and a logical restore. Figure 5-10 depicts a warm restore.



**Figure 5-10**  
*A Warm Restore*

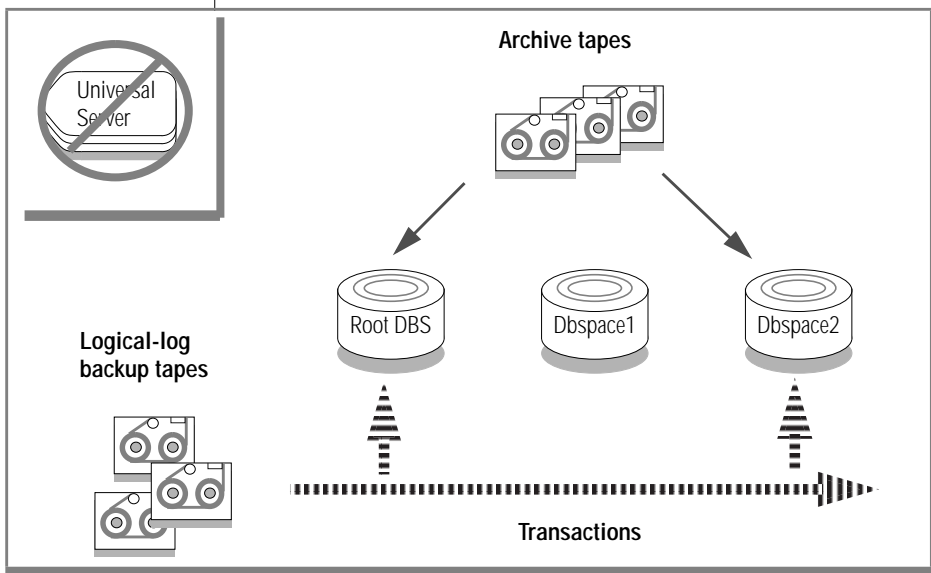
During a warm restore, backed-up logical-log files are replayed for the dbspaces that are being restored. To avoid overwriting the current logical log, the logical-log files to be replayed are written to temporary space. Therefore, a warm restore requires enough temporary space to hold the logical log (one set of logical-log files) or the number of log files being replayed, whichever is smaller. For information on how Universal Server looks for temporary space, see the discussion of **DBSPACETEMP** in the *INFORMIX-Universal Server Administrator's Guide*.



**Warning:** Make sure that you have enough temporary space for the logical-log portion of the warm restore; the maximum amount of temporary space that Universal Server needs is the size of the logical log (the size of all the logical-log files).

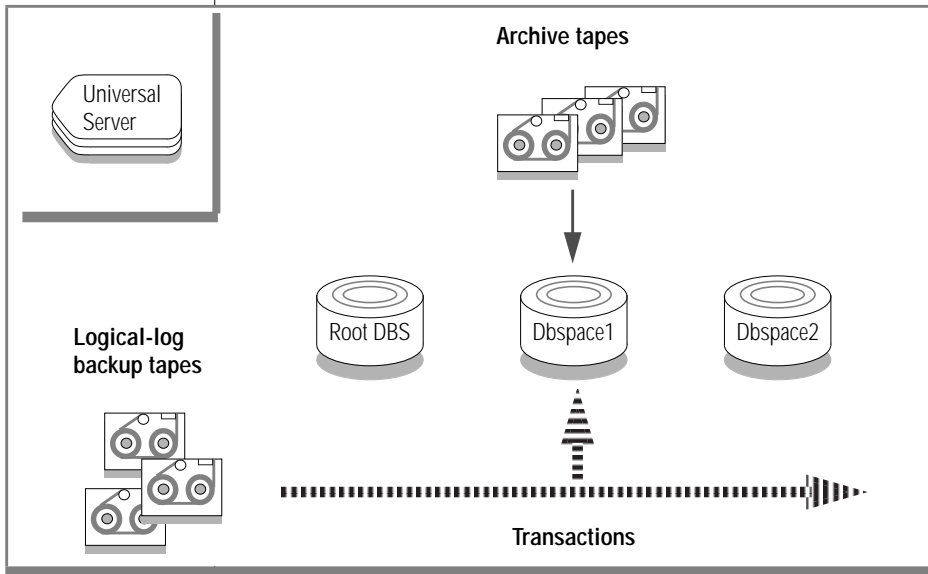
### A Mixed Restore

A *mixed restore* is a cold restore followed by a warm restore. A mixed restore restores some dbspaces and blobspaces during a cold restore (Universal Server is off-line) and some dbspaces and blobspaces during a warm restore (Universal Server is on-line). You might want to do a mixed restore if you are doing a full-system restore, but you need to provide access to a particular table or set of tables as soon as possible. In this case, you perform a cold restore to restore the critical dbspaces and the dbspaces that contain the important tables. Figure 5-11 illustrates the cold portion of a mixed restore.



**Figure 5-11**  
The Cold Portion of  
a Mixed Restore

Following the cold restore, you place Universal Server in on-line mode and perform a warm restore to restore the remaining dbspaces. Figure 5-12 illustrates the warm portion of a mixed restore.



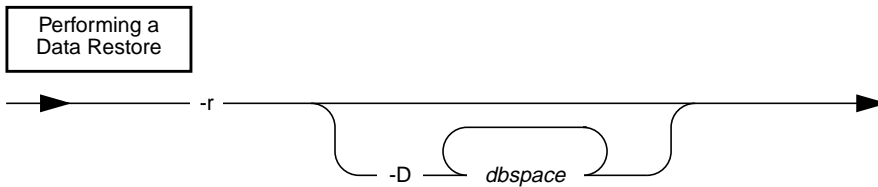
**Figure 5-12**  
*The Warm Portion  
of a Mixed Restore*

A cold restore takes less total time than a mixed restore to restore all your Universal Server data, even though Universal Server is on-line during part of a mixed restore. The mixed restore takes more time is because it requires two logical restores, one for the cold-restore portion and one for the warm-restore portion. A mixed restore, however, requires Universal Server to be *off-line* for less time than a cold restore.

The dbspaces that are not restored during the cold restore are not available until after they are restored during a warm restore, even though the failure to a critical dbspace might not have damaged them.

## Performing a Restore

Use the **ontape -r** option to perform a full physical and logical restore of Universal Server data. Use the **-D ontape** option to restore selected Universal Server dbspaces and blobspaces.



- r** directs **ontape** to perform a data restore (both physical and logical).
- D** directs **ontape** to restore only the dbspaces or blobspaces that you specify as part of a warm restore. Universal Server must be in on-line or quiescent mode to perform a warm restore.
- dbspace* is the name of a dbspace or blobspace to restore.

If you do not specify the **-D** option, **ontape** performs a full-system restore. Universal Server must be off-line to perform a full-system restore.

The **-r** option restores Universal Server data from the archive tape and the logical log backup tapes that you created since (and including) your last level-0 archive. If you use the **-D** option, you can restore selected dbspaces or blobspaces. For information, see “Steps to Restore Selected Dbspaces” on page 5-53.

### ***Steps to Restore the Whole System***

This section outlines the steps that you need to perform to restore your entire database server with **ontape**. The main steps in a full-system restore are as follows:

1. Gather the appropriate tapes.
2. Decide on a complete cold or a mixed restore.
3. Verify your Universal Server configuration.
4. Perform a cold restore.

Read these instructions and be familiar with them before you attempt a full-system restore.

#### *Gather the Appropriate Tapes*

You must gather the appropriate archive and logical-log tapes.

Before you start the restore, gather together all the archive tapes that you need. You need the tapes from your latest level-0 archive and the tapes from any subsequent level-1 or level-2 archives.

Identify the tape with the latest level-0 archive of the root dbspace. Use this tape first.

Gather together all the logical-log backup tapes since the latest level-0 archive.

#### *Decide on a Complete Cold or a Mixed Restore*

As mentioned in “Cold, Warm, or Mixed Restore—Choosing a Universal Server Mode” on page 5-44, when you are restoring your entire database server, you can restore the critical dbspaces (and any other dbspaces or blobspaces that you want to come on-line quickly) during a cold restore and then restore the remaining dbspaces and blobspaces during a warm restore. Decide before you start the restore if you want it to be completely cold or mixed.



### Verify Your Configuration

During a cold restore, you cannot reinitialize shared memory, add chunks, or change tape devices. When you begin the restore, the current Universal Server configuration must be compatible with, and accommodate, all parameter values that have been assigned since the time of the most-recent archive.

For guidance, use the copies of the configuration file that you create at the time of each archive. However, do not automatically set all current parameters to the same values as were recorded at the last archive. Pay attention to the following three groups of parameters:

- Shared-memory parameters

Verify that your current shared-memory parameters are set to the *maximum* value assigned since the level-0 archive. For example, if you decreased the value of `USERTHREADS` from 45 to 30 sometime since the level-0 archive, you must begin the restore with `USERTHREADS` set at 45, and not at 30, even though the configuration file copy for the last archive might have the value of `USERTHREADS` set at 30. (If you do not have a record of the maximum value of `USERTHREADS` since the level-0 archive, set the value as high as you think necessary. You might need to reassign values to `BUFFERS`, `LOCKS`, and `TBLSPACES` as well because the minimum values for these three parameters are based on the value of `USERTHREADS`.)

- Mirroring parameters

Verify that your current mirroring configuration matches the configuration that was in effect at the time of the last level-0 archive. Because Informix recommends that you create a level-0 archive after each change in your mirroring configuration, this step should not be a problem. The most critical parameters are the mirroring parameters, `MIRRORPATH` and `MIRROROFFSET`, which appear in the Universal Server configuration file.

- **Device parameters**

Verify that the files that have been used for Universal Server storage (of the dbspaces and blobspaces being restored) since the level-0 archive are available.

For example, if you dropped a dbspace or mirroring for a dbspace since your level-0 archive, you must ensure that the dbspace or mirror chunk device is available to Universal Server when you begin the restore. If Universal Server attempts to write to the chunk and cannot find it, the restore does not complete. Similarly, if you added a chunk since your last archive, you must ensure that the chunk device is available to Universal Server when it begins to roll forward the logical logs.

### *Perform a Cold Restore*

To perform a cold restore, Universal Server must be in off-line mode.

You must be a member of the Informix-Admin group to use **ontape**. Execute the following **ontape** command to restore all the dbspaces and blobspaces that Universal Server manages:

```
C:\> ontape -r
```

If you are performing a mixed restore, you restore only some of the dbspaces or blobspaces managed by Universal Server during the cold restore. You must restore at least all the critical dbspaces (the root dbspace and dbspaces with the physical-log and logical-log files), as follows:

```
C:\> ontape -r -D rootdbs llogdbs plogdbs
```

Before the restore starts, you are prompted to salvage the logical-log files on disk. Salvaging the log files is prudent; it saves log records that have not yet been backed up and enables you to recover your Universal Server data up to the point of the failure.

Use a new tape to salvage the log files.

During the restore, **ontape** prompts you to mount tapes with the appropriate dbspaces or log files.

If you are performing a mixed restore, you must restore all the logical-log files backed up since the last level-0 archive.

If you are performing a full restore, you have the option of not restoring logical-log files. If you do not back up your logical-log files, or choose not to restore them, you can restore your Universal Server data only up to its state at the time of your last archive. For more information, see “Do You Need to Back Up the Logical-Log Files?” on page 5-34.

At the end of the cold restore, Universal Server is in quiescent mode. You can bring Universal Server into on-line mode at this point and continue processing as usual.

If you restored only some of your dbspaces and blobspaces during the cold restore, you can start a warm restore of the remaining dbspaces and blobspaces after you bring Universal Server into on-line mode.

### ***Steps to Restore Selected Dbspaces***

This section outlines the steps that you must perform during a restore of selected dbspaces or blobspaces with **ontape** while Universal Server is in on-line or quiescent mode (a warm restore). The main steps in a warm restore are as follows:

1. Gather the appropriate tapes.
2. Verify your Universal Server configuration.
3. Back up logical-log files.
4. Perform a warm restore.

Read these instructions and be familiar with them before you attempt a restore.

#### *Gather the Appropriate Tapes*

You must gather the appropriate archive and logical-log tapes.

Before you start your restore, gather together all the tapes from your latest level-0 archive containing the dbspaces and blobspaces that you are restoring, and any subsequent level-1 or level-2 archives.

Gather together all the logical-log tapes from the logical-log backup prior to the latest level-0 archive of the dbspaces and blobspaces that you are restoring.

### *Verify Your Configuration*

During a warm restore, you do not need to worry about shared-memory parameters, as you do for cold restores.

Verify that the raw devices or files that have been used for Universal Server storage (of the dbspaces and blobspaces being restored) since the level-0 archive are available.

For example, if you dropped a dbspace or mirroring for a dbspace since your level-0 archive, you must ensure that the dbspace or mirror chunk device is available to Universal Server when you begin the restore. If Universal Server attempts to write to the chunk but cannot find the chunk, the restore does not complete. Similarly, if you added a chunk since your last archive, you must ensure that the chunk device is available to Universal Server when it begins to roll forward the logical logs.

### *Back Up Logical-Log Files*

Before you start a warm restore (even if you are performing the warm restore as part of a mixed restore), you must back up your logical-log files. See “Backing Up Logical-Log Files” on page 5-34.

After the warm restore, you *must* roll forward your logical-log files to bring the dbspaces that you are restoring to a state of consistency with the other dbspaces in the system. Failure to roll forward the logical log after you restore a selected dbspace results in the following message from **ontape**:

```
Partial system restore is incomplete.
```

### *Perform a Warm Restore*

To perform a warm restore, Universal Server must be in on-line or quiescent mode.

You must be a member of the Informix-Admin group to use **ontape**. Execute the **ontape** command, with the options shown in the following example, to restore selected dbspaces and blobspaces that Universal Server manages:

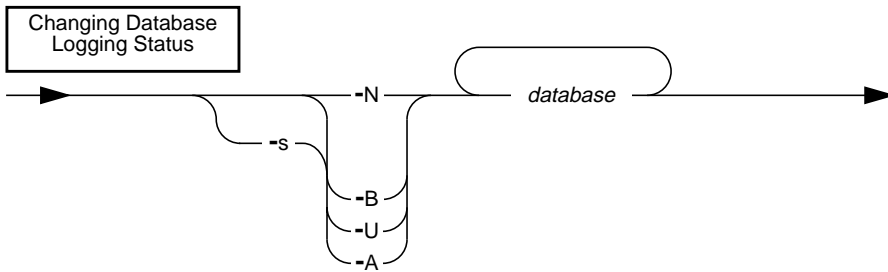
```
:> ontape -r -D dbspace1 dbspace2
```

You cannot restore critical dbspaces (the root dbspace and dbspaces with the physical log and logical-log files) during a warm restore; they must be restored as part of a cold restore, as described in “Steps to Restore the Whole System” on page 5-50.

During the restore, **ontape** prompts you to mount tapes with the appropriate dbspaces or log files.

At the end of the warm restore, the dbspaces or blobspaces that were down are on-line.

## Changing Database Logging Status



If you are adding logging to a database, you must create a level-0 archive before the change takes effect.

- A** directs **ontape** to change the status of the specified database to ANSI-compliant logging.
- B** directs **ontape** to change the status of the specified database to buffered logging.
- N** directs **ontape** to end logging for the specified database.
- s** initiates an archive.
- U** directs **ontape** to change the status of the specified database to unbuffered logging.
- database** is the name of the database. The database name cannot include a database server name.

For considerations about changing the logging status of a database, see the *INFORMIX-Universal Server Administrator's Guide* and “Scheduling Archives” on page 5-17.

# Summary of Universal Server for Windows NT

This appendix summarizes how Universal Server for Windows NT differs from Universal Server for UNIX as it is described in the *INFORMIX-Universal Server Administrator's Guide*.

The differences are organized into the following categories:

- Universal Server as a service
- Informix-Admin group
- Environment variables
- Pathnames and command-line prompt
- Communications
- Virtual processors
- Universal Server shared memory
- Disk space
- Configuration parameters
- Unsupported features

## Universal Server As a Service

Universal Server for Windows NT is installed as a Windows NT *service*. A service is roughly equivalent to a daemon process for the UNIX operating system. As a service, Universal Server can run in the background. It continues to run even if the current user logs off the computer and a new user logs on.

You must use the Services application in the Windows NT Control Panel to start and stop Universal Server. Using the Services application starts Universal Server under the user account **informix**.

For detailed descriptions of the procedures to start and stop Universal Server, see the *INFORMIX-Universal Server Installation Guide for Windows NT, Version 9.12*.

## Informix-Admin Group

The *INFORMIX-Universal Server Installation Guide for Windows NT, Version 9.12* does not use the user names **informix** and **root** for Universal Server administration because Windows NT does not support multiple login sessions for the same user. You must assign users who administer Universal Server to the Informix-Admin group. For an explanation of how to add a user to the Informix-Admin group, see the *INFORMIX-Universal Server Installation Guide for Windows NT, Version 9.12*.

## Environment Variables

The **INFORMIXSQLHOSTS** environment variable enables you to specify a particular computer for which the Windows NT registry serves as a central repository for the **sqlhosts** registry in the domain.



## Pathnames and Command-Line Prompt

Windows NT uses the backslash (\) to separate directory names within a command-line pathname. For example, the *INFORMIX-Universal Server Administrator's Guide* might show the pathname in the following example:

```
$ INFORMIXDIR/bin
```

The MS-DOS command-line form for Windows NT would be as follows:

```
%INFORMIXDIR%\bin
```

The DOS command-line prompt is typically as the following example shows:

```
C:\>
```

The *INFORMIX-Universal Server Administrator's Guide* shows the following command-line prompt:

```
%
```

However, you can use the Windows NT **prompt** command to set the command-line prompt to any symbol that you want.

## Communications

Universal Server for Windows NT supports the following communications protocols:

- TCP/IP for client-server and server-to-server communications using the Windows NT Sockets interface
- OSI TP4 transport for server-to-server communication

For more information about the communications facilities available for Universal Server for Windows NT, see Chapter 2, "Client/Server Communications."

## Virtual Processors

As described in Chapter 1, “Universal Server for Windows NT,” Universal Server virtual processors are implemented as Windows NT threads rather than as processes. Universal Server creates and manages Universal Server threads internally.

The following differences also exist with regard to virtual processors under Windows NT:

- Processor affinity is not supported.
- *Process aging* is not applicable under Windows NT.
- Only the SOC network virtual processor is supported.
- Kernel asynchronous I/O is used for all I/O except for miscellaneous low-priority system I/O.
- The OPT (optical disk) virtual processor is not supported.

## Universal Server Shared Memory

The following changes affect Universal Server shared memory for Windows NT:

- SHMBASE and SHMADD must be multiples of 64 kilobytes.
- The communications portion of shared memory is not used.
- Page size is 4 kilobytes.

## Disk Space

Universal Server for Windows NT uses only NTFS files for disk space. Universal Server for Windows NT does not use raw disk space.

Mirroring is implemented as the *INFORMIX-Universal Server Administrator's Guide* describes.

## Configuration Parameters

For the default values of Universal Server configuration parameters for Windows NT, see the **onconfig.std** file in the %INFORMIXDIR%\etc directory.

The following Universal Server configuration parameters are *not* supported for Windows NT:

- ADTPATH
- ADTSIZE
- AFF\_NPROCS
- AFF\_SPROC
- CONSOLE
- DUMPCNT
- DUMPCORE
- DUMPSHMEM
- NOAGE
- STAGEBLOB

The default values of the following configuration parameters are also significant:

- DBSERVERNAME is set during the installation.
- MSGPATH is set to %INFORMIXDIR%\online.log.
- SERVERNUM is ignored. This parameter has no significance because multiple residency (multiple Universal Server instances on the same computer) is not supported.
- SHMBASE and SHMADD values must be multiples of 64 kilobytes.

## **Unsupported Features**

Universal Server for Windows NT does not support the following features. Ignore any references to these features in the documentation set:

- Multiple residency, or multiple instances of Universal Server on the same Windows NT server
- Use of raw disk space
- Shared-memory as a local client-server communications interface
- ON-Archive backup and recovery system

### ***Universal Server Environment Variables***

The environment variables **ARC\_DEFAULT** and **ARC\_KEYPAD** are not supported because ON-Archive is not supported.

### ***Files***

Universal Server for Windows NT does not use the following files:

- **ARCreqid.NOT**
- **config.arc**
- **oncatlgr.out.pidnum**
- **oper\_deflt.arc**
- **status\_vset\_volnum.itgr**
- **ssyfail.pidnum**
- **core**
- **gcore**
- **tctermcap**

---

# Using OSI TP4

Universal Server for Windows NT supports the OSI TP4 transport for server-to-server communication.

---

## Implementing OSI TP4 for Universal Server

To use the OSI TP4 transport for server-to-server communication, you must install and configure both an appropriate adapter card and the OSI TP4 network software. Use the Network option in the Windows NT Control Panel to install and configure the OSI TP4 network software. Perform the following steps:

1. Click **Add Software**.
2. From the Network Software window, choose **<Other> Requires disk from Manufacturer**.
3. Insert the disk that contains the OSI TP4 network software into the drive of your choice and enter the appropriate drive letter and path in the Insert disk window.
4. Select **ISO TP4/CLNP Stack Network Software Component**.
5. Click **OK**.

During the installation, be sure to note the name that you assign as the Network Service Access Point (NSAP) name. The name of the computer is the default value for the NSAP name. You need to supply the name that you select as a value in the entries that you create in the **osirte** file and in the Universal Server **sqlhosts** registry. For information about the values that you can assign as the NSAP name, see the OSI TP4 documentation.

## The osirte File

After you install the OSI TP4 network software, you must create an entry in the **osirte** file in the %SYSTEMROOT%\drivers\etc directory for each Windows NT server with which this server will communicate. The **osirte** file is the OSI TP4 routing table. The format of an entry in the **osirte** file is as follows:

```
NSAPname:$net_addr:priority:net_card
```

The values for an entry in the **osirte** file are as follows:

- *NSAPname* is the NSAP name that you assigned when you installed the OSI TP4 network software.
- The value for *net\_addr* is the Ethernet or X.121 address for the machine in hexadecimal.
- The value for *priority* is 0 or 1.
- The value for *net\_card* is the device name of the network card.

The following example illustrates an entry in the **osirte** file:

```
WARSPITE:$00001B15B023:1:\Device\NE20001
```

The symbol \$ at the beginning of the first or second field (*NSAPname* and *net\_addr*, respectively) indicates that a hexadecimal value follows.

To obtain the values of the network address and the network device name, use the Windows NT **Diagnostics** application under **Administrative Tools**. To display this information, click the **Network . . .** button.

## The sqlhosts Registry

You must also create an entry in the Universal Server **sqlhosts** registry for OSI TP4. The format of the entry in the **sqlhosts** registry is as follows.

Registry Key	PROTOCOL	HOST	SERVICE
<i>Universal Server name</i>	onsocosi	WARSPITE	12

For an OSI TP4 entry, the **PROTOCOL**, **HOST**, and **SERVICE** fields have the following requirements:

- The value of the **nettype** field must be `onsocosi`.
- The value of the **hostname** field must be *exactly* the same as the NSAP name that you specified when you installed the network software. You must enter uppercase characters as uppercase characters and lowercase characters as lowercase characters.
- The value of the **servicename** field is the OSI transport selector number. The number that you assign is an arbitrary number, but you must ensure that it is a unique OSI transport selector number within the network.





# Index

## A

Access method  
     secondary 4-48  
 Access method name  
     ANSI-compliant 4-51  
 Access method, secondary 4-48,  
     4-49  
 Administering database server,  
     Informix-Admin Group A-2  
 Algorithm, in-place alter 3-44, 4-20  
 ALTER FRAGMENT statement 3-4,  
     3-45  
 ALTER INDEX statement 3-32,  
     3-43, 3-45  
 ALTER TABLE statement 3-37, 3-42  
     in-place alter algorithm 3-44, 4-20  
     to add or drop a column 3-44  
     to change data type 3-44  
     to change extent sizes 3-37  
 AND keyword 4-46  
 ANSI-compliant database  
     access method name 4-51  
 Archive  
     and one device 5-11  
     before you create 5-21  
     creating 5-24  
     criteria for disk pages 5-31  
     description of 5-4  
     details of 5-27  
     disk or tape, where to store 5-4  
     if device is /NUL 5-32  
     if interrupted 5-26  
     if logical log fills 5-25  
     labelling the tape 5-23  
     levels 5-15  
     log space required 5-28  
     scheduling 5-17

Archive and restore  
     fragmentation for 3-59  
 Assigning table to a dbspace 3-4  
 Attached index 3-39  
 Audit configuration  
     changing from a command  
         line 1-15  
     showing from a command  
         line 1-15  
     showing with onshowaudit 1-15  
 Audit error mode, and onaudit 1-16  
 Audit trail, extracting information  
     with onshowaudit 1-17  
 Auditing  
     and audit trail files 1-13  
     and ONCONFIG file 1-16  
     changing the configuration 1-15  
     description of 1-12  
     error mode levels 1-16  
     message server 1-12  
     showing configuration 1-15  
     table for SQL 1-18  
     turning off with onaudit 1-16  
     turning on with onaudit 1-16

## B

Backup logical-log, description  
     of 5-5  
 Balanced B-tree  
     description of 3-15  
 Blobpages  
     during archive 5-40  
     estimating number in  
         tblspace 3-27  
 Blobspace  
     creating 1-7  
     new, when available 5-35

- Block size
  - and tape device 5-13
  - parameter 5-13
- Branch index pages 3-14, 3-20
- B-tree index
  - balanced 3-15
  - btree cleaner 4-42
  - cleaner 3-30
  - GROUP BY clause 4-38
  - ORDER BY clause 4-37, 4-38
  - structure 3-14 to 3-16
  - structure of leaf node entries 3-15
  - usage 4-48
- BYTE data type 3-9, 3-28, 3-29

---

## C

- Casting
  - explicit, definition of 4-21
  - implicit, definition of 4-21
- Central registry, sqlhosts 2-11
- Changing the system audit configuration 1-15
- CHAR data type 3-47, 4-22
- Choosing distribution scheme 3-61
- Chunk
  - adding 1-7
  - allocating initial 1-7
- Chunks
  - monitoring 3-41
- Client-server protocol 2-5
- Clustering 3-32, 3-43, 4-39
- Cold restore
  - and mixed restore 5-47
  - description of 5-44
  - performing with ontape 5-52
- Column filter expression 4-25
- Columns
  - filtered 4-36
  - order-by and group-by 4-37
  - with duplicate keys 4-38
- Command-line prompt, Windows NT A-3
- Communications
  - shared-memory A-4
  - supported protocols A-3
- Composite index
  - order of columns 4-40
  - use 4-39
- Concurrency, effects of isolation level 4-13
- Configuration parameters
  - DBSPACETEMP 3-7
  - DS\_TOTAL\_MEMORY 4-72
  - for virtual processors 1-6
  - MAX\_PDQPRIORITY 4-69
  - not supported A-5
  - onconfig.std file A-4
  - PDQPRIORITY 4-71
  - SHMADD A-5
  - SHMBASE A-5
  - unsupported A-5
- Configuration parameters, ontape
  - checking 5-14
  - setting 5-9
- Configuring
  - network card 2-6
  - TCP/IP network software 2-6
- CONNECT statement 3-4
- Connecting to the Windows NT Network application 2-6
  - creating a TCP/ IP connection 2-6
  - from different types of clients 2-5
  - methods of 2-5
  - server to server 2-5
  - sqlhosts registry 2-8
  - TCP/IP, steps 2-6
  - types of clients 2-4
- Contention
  - disk 4-17
  - reducing through fragmentation 3-57
- Contiguous extents
  - performance advantage 3-42
- Conventions Intro-6
- Correlated subquery
  - description of 4-31
  - rewritten as join 4-31
  - rewritten as separate queries 4-32
- CREATE CLUSTERED INDEX statement 3-32
- CREATE INDEX statement 3-43, 3-44, 4-49
  - USING clause 4-52
- CREATE TABLE statement 3-4, 3-28, 3-37
  - USING clause 4-63

---

## D

- Data distributions
  - creating on filtered columns 4-33
  - description 4-7
  - use with optimizer 4-7
- Data restore, minimizing time needed 5-20
- Data type
  - built-in 4-48
  - distinct 4-47
  - opaque 4-47
- Data types
  - BYTE 3-9, 3-28, 3-29
  - CHAR 3-47, 4-22
  - effect of mismatch 4-20
  - NCHAR 3-47
  - NVARCHAR 3-12, 3-47
  - TEXT 3-9, 3-28, 3-29, 3-47
  - VARCHAR 3-12, 3-47, 4-22
- DATABASE statement 3-4
- Database-logging status
  - changing 5-55
- DB-Access utility 3-42
- dbload utility 3-34, 3-42
  - table for audit data 1-18
- dbschema utility 3-55, 3-62, 4-33
- DBSERVERNAME parameter
  - default value A-5
- Dbspaces
  - archiving 5-4
  - creating 1-7
  - for temporary tables and sort files 3-7
  - multiple disks within 3-6
  - reorganizing to prevent extent interleaving 3-42
  - restore selected 5-44
  - restoring 5-7
- DBSPACETEMP
  - environment variable 3-7
  - Universal Server parameter 3-7
- Decision-support queries (DSS) 3-53, 4-77
- Demonstration database Intro-5
- Denormalizing data model 3-46
- Denormalizing tables 3-46
- Derived data 3-50

- Disk access
  - effect of contention 4-17
  - indexed 4-19
  - latency of 4-18
  - nonsequential 4-19
  - performance 4-45, 4-46
  - sequential 4-18, 4-45, 4-46
  - sequential forced by query 4-32, 4-33, 4-46
- Disk extent 3-36
- Disk layout
  - and archiving 3-8
  - and table isolation 3-5
- Disk space, allocating 1-7
- Disks, multiple within dbspace 3-6
- Displaying, audit
  - configuration 1-15
- DISTINCT keyword 4-40
- Distribution scheme 3-60
- DISTRIBUTIONS ONLY
  - clause 4-34
- Documentation notes Intro-13
- Domain Name Server 2-6
- Domain, Windows NT
  - and database server 2-3
  - and database server user
    - accounts 2-6
  - and user accounts 2-3
  - controller 2-3
  - description of 2-3
  - resolving names 2-6
  - trusted 2-3
- DROP INDEX statement, releasing an index 4-45
- Dropping indexes 3-33
- DS\_MAX\_QUERIES Universal
  - Server parameter 4-66, 4-74
- DS\_MAX\_SCANS Universal Server
  - parameter 4-66, 4-73
- DS\_TOTAL\_MEMORY database
  - server parameter 4-72
- DS\_TOTAL\_MEMORY Universal
  - Server parameter 4-68
- Duplicate index keys 4-38
- Dynamic-index join 4-12

## E

- Eliminating fragments from
  - scans 3-54
- Environment variables
  - DBSPACETEMP 3-7
  - INFORMIXSQLHOSTS A-2
  - OPTCOMPIND 4-75
  - PDQPRIORITY 4-67, 4-68
  - PSORT\_DBTEMP 3-7
  - unsupported A-6
- equal() function 4-59
- Error message files Intro-12
- Error mode, and onaudit 1-16
- Event logs, how used 1-11, 1-13
- Explicit casting, definition of 4-21
- Expression-based distribution
  - scheme
    - designing 3-62
    - which type to use 3-62
- EXTENT SIZE clause 3-37
- Extents
  - eliminating interleaved 3-41
  - index size 3-39
  - interleaved 3-41
  - managing 3-36
  - monitoring 3-41
  - next extent size 3-37, 3-38
  - performance 3-41
  - reclaiming empty space 3-44
  - reorganizing dbspace to prevent interleaving 3-42
  - size of 3-37, 3-39

## F

- Features, unsupported A-6
- Files
  - not used A-6
  - sort 3-7
- FILLFACTOR Universal Server
  - parameter 3-18
- Filter
  - columns 4-25
  - introduced 4-8
  - selectivity 4-9, 4-31

- Filter expression
  - effect on performance 4-24, 4-32, 4-33
  - evaluated from index 4-39
  - join without 4-23
  - optimizer uses 4-24
  - selectivity estimates 4-9, 4-31
- Filtered columns 4-36
- Formula
  - B-tree index pages 3-18
  - decision support queries 4-72
  - index extent size 3-39
  - index pages 3-11
  - memory grant basis 4-73
  - number of remainder pages 3-10
  - partial remainder pages 3-11
  - quantum of memory 4-66
  - rows per page 3-10
  - R-tree index pages 3-23
  - sbpages 3-25
  - sbspace metadata 3-25
  - scan threads 4-67
  - scan threads per query 4-73
  - simple-large-object pages 3-26, 3-27
  - sort operation, costs of 4-16
- FRAGMENT BY EXPRESSION
  - clause 3-64
- Fragment ID 3-52
- Fragmentation
  - determining strategy 3-55
  - distribution scheme 3-60
  - for finer granularity of archive and restore 3-59
  - for increased availability of data 3-58
  - for parallel I/O 3-56
  - goals 3-56
  - guidelines 3-51
  - improving 3-65
  - monitoring 3-64
  - of indexes 3-64
  - setting priority levels for PDQ 4-75
  - strategy 3-54
  - to reduce contention 3-57

Functional index  
  creating 4-55 to 4-56  
  definition 4-37  
  usage 4-54  
Functions  
  indexing 4-37

---

## G

Global Language Support  
  (GLS) 4-20, 4-22  
greaterthanorequal() function 4-59  
greaterthan() function 4-59  
GROUP BY clause 4-6  
  composite index used for 4-39  
  indexes for 4-43  
Group-by columns 4-37

---

## H

Hash join 4-12, 4-28  
  and isolation level 4-13  
HKEY\_CLASSES\_ROOT  
  subtree 1-9  
HKEY\_CURRENT\_USER  
  subtree 1-9  
HKEY\_LOCAL\_MACHINE  
  subtree 1-9  
HKEY\_USERS subtree 1-9  
Home pages, in indexes 3-10  
Horizontal fragmentation 3-51  
hosts file, creating an entry 2-6

---

## I

Implicit casting, definition of 4-21  
Improving fragmentation 3-65  
IN DBSPACE clause 3-4  
Indexes  
  adding for performance 4-10, 4-34  
  and filtered columns 4-36  
  avoiding duplicate keys 4-38  
  B-tree cleaner 3-30, 4-42  
  column 4-54  
  composite 4-39  
  cost of access using 4-19  
  disk space used by 3-31, 4-45

  dropping 3-33  
  duplicate entries 4-38  
  effect of physical order 4-27  
  effect of updating 3-30, 4-42  
  estimating pages 3-16, 3-22 to 3-24  
  extent size 3-39  
  fragmenting 3-64  
  functional 4-54  
  impacts on delete, insert, and  
    update operations 3-31  
  managing 3-29  
  next-extent size 3-39  
  order-by and group-by  
    columns 4-37  
  ordering columns in  
    composite 4-40  
  scan, key-only 4-11  
  space costs 3-30  
  subquery use of 4-31  
  time cost of 3-31  
  used by optimizer 4-26  
  when not used by optimizer 4-21,  
    4-32, 4-33, 4-46  
  when to rebuild 4-42  
Informix-Admin group  
  description of 1-6  
  to administer database server A-2  
INFORMIXSQLHOSTS  
  environment variable 2-11  
informix, user  
  and database server 1-6, A-2  
  starting and stopping database  
    server 1-8, A-2  
Input-output  
  contention and high-use tables 3-5  
  parallel for individual  
    queries 3-56  
INSERT cursor 3-60  
INSERT statement 3-60  
Installing, TCP/IP network  
  software 2-6  
Interleaved extents 3-41  
Internet domain addresses,  
  resolving 2-6  
Isolating tables 3-5  
Isolation levels  
  Repeatable Read 4-11, 4-13

---

## J

Join  
  dynamic-index 4-12  
  effect of large join on  
    optimization 4-47  
  hash 4-12, 4-28  
  nested-loop 4-12  
  ordered pair 4-12  
  pair 4-6  
  sort-merge 4-13, 4-28  
  with column filters 4-25  
  without filters 4-23  
Join and sort, reducing impact  
  of 4-42  
Join method  
  hash 4-5, 4-12, 4-13, 4-66, 4-75  
  influencing with  
    OPTCOMPIND 4-12  
  sort-merge 4-5, 4-13, 4-28, 4-75

---

## K

Kernel asynchronous I/O  
  and virtual processors A-4  
  when used 1-5  
Key-only index scan 4-11

---

## L

Large tables  
  filtered columns 4-36  
Latency 4-18  
Leaf index pages 3-14, 3-20  
lessthanorequal() function 4-59  
lessthan() function 4-59  
LIKE test 4-32  
LOAD and UNLOAD  
  statements 3-4, 3-34, 3-41, 3-42,  
    3-45  
Local loopback connection  
  and sqlhosts registry 2-9  
  description of 2-5  
  for local database server client 2-5  
Locale, assumptions about Intro-4  
LocalSystem user account  
  and event logs 1-11

Locating simple-large-object data 3-28

Logical log

- automatic backup, starting 5-36
- backing up 5-34
- backup, and separate devices 5-11
- backup, description of 5-5
- backup, if fills during archive 5-25
- continuous, starting 5-37
- files, backing up 5-34
- files, backup criteria for blobpages 5-40
- files, importance of backing up 5-34
- files, salvaging 5-6
- files, used status 5-36

Logical restore

- and cold restore 5-44
- and warm restore 5-46
- description of 5-7

LTAPEBLK parameter, description 5-9

LTAPEDEV parameter

- changing to /NUL 5-12
- description 5-9
- if two tape devices 5-25

LTAPESIZE parameter

- description 5-9
- setting 5-13

---

## M

Managing extents 3-36

MAX\_PDQPRIORITY

- limiting PDQ priority 4-68

MAX\_PDQPRIORITY Universal

- Server parameter 4-69

Memory Grant Manager (MGM) 4-65

Message Server, database server 1-12

Mixed restore, description of 5-47

MODIFY NEXT SIZE clause 3-37

MS-DOS, command-line prompt A-3

MSGPATH default value A-5

Multiple residency A-5

---

## N

NCHAR data type 3-47

Nested-loop join 4-12

Network

- address, obtaining B-2
- client, and sqlhosts registry 2-9
- creating a TCP/IP connection 2-6
- device name, obtaining B-2
- interface card, configuring 2-6
- performance of 4-22
- TCP/IP connection types 2-5

NEXT SIZE clause 3-37

Next-extent size

- index 3-39

Nonsequential disk access 4-19

NSAP name

- and sqlhosts file entry B-3
- assigning B-2

NTFS files, database server chunks 1-7, A-4

null file

- for creating a chunk 1-7

NUL, ontape

- as a tape device 5-12

NVARCHAR data type 3-12, 3-47

---

## O

onaudit utility

- auditing mode levels 1-16
- changing the system audit configuration 1-15
- description of 1-14
- error mode levels 1-16
- options 1-12
- showing the auditing configuration 1-15
- syntax 1-14

oncheck utility 3-9, 3-18, 3-38

- and index sizing 3-18
- determining free space in index 4-42
- pe option 3-41
- pr option 3-9, 3-25, 3-40
- sbospace sizing 3-27

ONCONFIG file, and audit configuration 1-16

onconfig.std file A-4

onload and onunload utilities 3-4, 3-45

onmode utility, -MQDS

- options 4-69

onshowaudit utility

- description of 1-17
- format of data file for fragmented tables 1-18
- options 1-13
- syntax 1-17
- who can run 1-17

onstat utility

- c option 3-27
- g mgm option 4-66
- g ppf option 3-64
- t option 3-7

ontape utility

- archive levels 5-15
- archives and database server modes 5-22
- archive, details of 5-27
- archive, estimating time for 5-18
- backing up logical log 5-34
- before creating archive 5-17
- changing database logging status 5-55
- changing LTAPEDEV to /dev/null 5-12
- changing parameters using an editor 5-10
- changing TAPPEDEV to /dev/null 5-12
- checking configuration parameters 5-14
- configuration parameters 5-9
- creating an archive 5-15
- example 5-25
- exit codes 5-15
- if archive terminates prematurely 5-22
- labelling archive tapes 5-22
- logical-log backup, details of 5-38

- option
  - a 5-36
  - c 5-37
  - D 5-49
  - L 5-24
  - r 5-49
  - s 5-24
- physical restore, choosing
  - type 5-43
- precautions, one tape device 5-11
- restore, choosing database server
  - mode 5-44
- restoring data 5-43
- scheduling archives 5-17
- starting continuous backup 5-37
- syntax 5-14
- syntax, logical-log file
  - backup 5-36
- Operator class
  - definition 4-50, 4-57
- OPTCOMPIND
  - and Repeatable Read 4-12
  - environment variable 4-75
  - Universal Server parameter 4-11, 4-75
- Optimizer
  - and hash join 4-28
  - and OPTCOMPIND 4-12, 4-13
  - and SET OPTIMIZATION
    - statement 4-47
  - and sort-merge join 4-28
  - autoindex path 4-39
  - composite index use 4-39
  - data distributions 4-33
  - filter selectivity 4-9, 4-31
  - index not used by 4-32, 4-46
  - index used by 4-7
  - introduced 4-5
  - specifying high or low level of
    - optimization 4-47
  - system catalog use 4-7
  - when index not used 4-33
- ORDER BY clause 4-6, 4-43
- Order-by columns 4-37
- OSI TP4
  - and server to server
    - communication A-3, B-1
  - osirte file B-2

- osirte file
  - for OSI TP4 B-2
  - format of B-2

---

## P

- Page buffer
  - effect on performance 4-17
  - restrictions with simple-large-object data 3-29
- Page size A-4
- Parallel database queries (PDQ)
  - allocating resources for 4-67
  - and SQL 3-51
  - and the SET PDQPRIORITY
    - statement 4-75
  - controlling resources for 4-76
- Parallel sorts
  - interaction with PDQ 4-43
- Partitioning 3-51
- Password
  - and the CONNECT statement 2-7
  - and the hosts.equiv file 2-7
  - .netrc file 2-7
- Pathnames, Windows NT A-3
- PDQPRIORITY
  - configuration parameter 4-71
  - environment variable 4-67, 4-68
  - limiting PDQ priority 4-68
- Performance
  - contiguous extents 3-41
  - disk access 4-45, 4-46
  - disk latency 4-18
  - dropping indexes to speed
    - modifications 3-33
  - effect of
    - correlated subquery 4-31
    - data mismatch 4-21
    - duplicate keys 4-38
    - filter expression 4-24, 4-32, 4-33
    - indexes 4-34 to 4-42
    - regular expressions 4-32
    - simple-large-object location 3-28
    - table size 4-45
  - filter selectivity 4-9, 4-31

- improved by
  - specifying optimization
    - level 4-47
  - temporary table 4-44
- index time during
  - modification 3-31
- seek time 4-18
- sequential access 4-45, 4-46
- slowed by duplicate keys 4-38
- use of redundant data 3-50
- Physical restore
  - and cold restore 5-44
  - data restored 5-8
  - description of 5-7
  - types of 5-8
- Process aging A-4
- Processor affinity A-4
- prompt command A-3
- PROTOCOL field in sqlhosts
  - registry 2-10
- PSORT\_DBTEMP environment
  - variable 3-7

---

## Q

- Queries
  - decision-support 3-53, 4-77
  - improving performance of 4-29
- Query optimizer
  - and hash join 4-28
  - and sort-merge join 4-28
  - display query plan 4-14
  - introduced 4-5
- Query plan
  - autoindex path 4-39
  - description 4-5
  - indexes in 4-26
  - row access cost 4-17
  - selection 4-6
  - time costs of 4-15
  - using indexes 4-27

---

## R

- Raw disk space 1-7, A-4
- Reclaiming empty extent space 3-44
- Redundant data, introduced for
  - performance 3-50
- Redundant pairs 4-6
- regedt32 program
  - and sqlhosts registry 2-12
  - description of 1-9
- Registry
  - sqlhosts 2-8
- Registry, Windows NT
  - advantages of 1-10
  - and sqlhosts registry 2-8
  - description of 1-9
  - how database server uses 1-10
- Regular expression, effect on
  - performance 4-32
- Relational model,
  - denormalizing 3-46
- Release notes Intro-13
- Remainder pages 3-10
- Repeatable Read isolation level 4-11
- Restore
  - full-system 5-43
  - physical, data restored 5-8
  - physical, types of 5-8
  - selected dbspaces 5-44
  - selected dbspaces, steps 5-53
  - whole system, steps 5-50
- Restore, database server
  - description of 5-7
  - logical, description of 5-7
  - physical, description of 5-7
- Root index page 3-14, 3-20
- root user 1-6, A-2
- Round-robin distribution
  - scheme 3-61 to 3-62
- Row access cost 4-17
- R-tree index
  - bounding box 3-19
  - description 3-19
  - estimating size 3-22 to 3-24
  - structure of leaf node entries 3-21
  - usage 4-48

---

## S

- Salvaging logical-log files 5-6
- Sbpages
  - estimating number in
    - sbspace 3-25
- Sbspace
  - estimating size 3-25
  - monitoring metadata space 3-46
- Scans, limiting number of 4-73
- Secondary access method 4-48
  - defined by server 4-49
  - definition 4-48
  - description of 4-49
- Seek time 4-18
- SELECT statement 3-9, 4-6, 4-23, 4-25, 4-26
- Selectivity
  - and indexed columns 4-38
  - column, and filters 4-36
  - of a query 4-8, 4-30
- Sequential access 4-18
- Sequential scans 4-45
- SERVERNUM parameter A-5
- Server-to-server communication
  - OSI TP4 A-3
  - using OSI TP4 B-1
- Server-to-server connections
  - TCP/IP 2-5
- Services application, Windows NT 1-8
  - starting and stopping database server A-2
- SET DATASKIP statement 3-58
- SET EXPLAIN statement 3-55, 4-14
- SET OPTIMIZATION statement 4-47
- SET PDQPRIORITY statement 4-68, 4-71, 4-75
- setup program
  - and sqlhosts registry 2-8
- Shared memory
  - database server A-4
  - for communications A-4, A-6
  - page size A-4
- SHMADD configuration
  - parameter A-4, A-5

- SHMBASE configuration
  - parameter A-5
- SHMBASE parameter A-4
- Showing auditing
  - configuration 1-15
- Simple large object 3-28
- Smart large objects
  - estimating sbspace size 3-25
  - fragmentation 3-60
  - fragmenting 3-52
  - improving I/O 3-6
  - monitoring metadata space 3-46
- Sockets interface (UNIX) 2-10
- Sorting
  - avoiding with temporary table 4-44
  - effect on performance 4-42
  - time costs of 4-16
- Sort-merge join 4-13, 4-28
  - and isolation level 4-13
- SQLCODE field of SQL Communications Area 3-48
- Sqlhosts registry
  - and INFORMIXSQLHOSTS environment variable A-2
  - and local loopback connection 2-9
  - and network client 2-9
  - and Windows NT registry 2-8
  - central registry 2-11
  - description of 2-8
  - entry for OSI TP4 B-3
  - entry for TCP/IP 2-8
  - how it is created 2-8
  - INFORMIXSQLHOSTS environment variable 2-11
  - location of 2-11
  - PROTOCOL field 2-10
  - steps to change 2-12
  - values in 2-8
- Starting and stopping database server
  - how to 1-8
  - if it fails to start 1-8
- Statistics
  - choosing a query plan 4-6
  - in system catalog tables 4-8
- Strategy function
  - description of 4-58

- Strings, expelling long 3-47
- Structured Query Language (SQL)
  - ALTER FRAGMENT
    - statement 3-4, 3-45
  - ALTER INDEX statement 3-32, 3-43, 3-45
  - ALTER TABLE statement 3-37, 3-42, 3-44
  - AND keyword 4-46
  - CONNECT statement 3-4
  - CREATE CLUSTERED INDEX
    - statement 3-32
  - CREATE INDEX statement 3-43, 3-44
  - CREATE TABLE statement 3-4, 3-28, 3-37
  - DATABASE statement 3-4
  - DISTINCT keyword 4-40
  - DISTRIBUTIONS ONLY
    - clause 4-34
  - EXTENT SIZE clause 3-37
  - FRAGMENT BY EXPRESSION
    - clause 3-64
  - GROUP BY clause 4-6
  - IN DBSPACE clause 3-4
  - INSERT statement 3-60
  - LOAD and UNLOAD
    - statements 3-4, 3-34, 3-41, 3-42, 3-45
  - MODIFY NEXT SIZE clause 3-37
  - NEXT SIZE clause 3-37
  - ORDER BY clause 4-6
  - SELECT statement 3-9, 4-6, 4-23, 4-25, 4-26
  - SET DATASKIP statement 3-58
  - SET EXPLAIN statement 3-55, 4-14
  - SET OPTIMIZATION
    - statement 4-47
  - SET PDQPRIORITY
    - statement 4-68, 4-71, 4-75
  - TO CLUSTER clause 3-43, 3-45
  - UNION keyword 4-46
  - UPDATE STATISTICS
    - statement 4-6, 4-7, 4-33
  - WHERE clause 4-30, 4-32, 4-33, 4-37, 4-46

- Subquery
  - correlated 4-31
  - performance of 4-31
- Subtrees
  - of Windows NT registry 1-9
- Support function
  - description for secondary access
    - method 4-58
- suserv.log file 1-8
- Symbol table 3-48
- Syntax
  - onaudit utility 1-14
  - onshowaudit utility 1-17
- sysams system catalog table 4-50, 4-61
- syscolumns system catalog
  - table 4-33
- sysdistrib system catalog table 4-33
- System catalog tables 4-8
- system.ini file 1-9

---

## T

- Table partitioning 3-51
- Table size
  - calculating 3-8
  - with variable-length rows 3-11
- Tables
  - adding redundant data 3-50
  - assigning to dbspace 3-4
  - companion, for long strings 3-47
  - cost of access 4-45
  - cost of in-place alter table 4-20
  - costs of companion 3-49
  - denormalizing 3-46
  - division by bulk 3-49
  - estimating
    - blobpages in tblspace 3-27
    - data pages 3-9
    - index pages 3-13
    - sbpages in sbspace 3-25
    - size with fixed-length rows 3-9
    - size with variable-length
      - rows 3-11
  - expelling long strings 3-47
- fragmentation
  - guidelines 3-51
  - strategy 3-54
- frequently updated
  - attributes 3-49
- infrequently accessed
  - attributes 3-49
- isolating high-use 3-5
- managing
  - extents 3-36
  - indexes 3-29
- multiple access arms to reduce
  - contention 3-6
- nonfragmented 3-8
- order and performance 4-23
- placement on disk 3-3
- reducing contention between 3-5
- redundant and derived data 3-50
- rows too wide 3-48
- shorter rows 3-46
- temporary 3-7
- variable-length rows 3-11

Tape

- gathering for cold restore 5-50
- gathering for warm restore 5-53
- ontape block-size parameters 5-13

Tape device

- and block size 5-13
- precautions with only one 5-11

Tape device, ontape

- parameters, setting 5-10
- using NUL 5-12

TAPEBLK parameter,
 

- description 5-9

TAPEDEV parameter

- changing to /NUL 5-12
- description of 5-9
- if the device is NUL 5-32
- if two tape devices 5-25

TAPESIZE parameter,
 

- description 5-9

Tblspace 3-8

Tblspace ID 3-52

TCP/IP

- for client-server
  - communication 2-5
- for database server A-3
- network software 2-6



- TCP/IP connection
  - and sqlhosts registry 2-9
  - and Windows sockets 2-5
- Temporary tables 4-44
- TEXT data type 3-9, 3-28, 3-29, 3-47
- Threads
  - database server A-4
  - Windows NT 1-3, A-4
- Threads per query 4-72
- Thread-safe virtual processor 1-5
- TO CLUSTER clause 3-43, 3-45
- Transport Layer Interface (UNIX) 2-10
- Trusted clients, database server and
  - Windows NT domains 2-4
  - hosts.equiv file 2-7
- Trusted domain 2-3

---

## U

- UNION keyword 4-46
- Universal Server
  - as a service A-2
  - if fails to start 1-8
  - keys in Windows NT registry 1-10
  - running in background A-2
  - shared memory A-4
  - starting and stopping 1-8
  - threads A-4
- Universal Server parameters
  - DS\_MAX\_QUERIES 4-66, 4-74
  - DS\_MAX\_SCANS 4-66, 4-73
  - DS\_TOTAL\_MEMORY 4-68
  - FILLFACTOR 3-18
  - OPTCOMPIND 4-11, 4-75
- UPDATE STATISTICS
  - statement 4-6, 4-7, 4-33
- User accounts and Windows NT domains 2-3
- User authentication
  - CONNECT statement 2-7
  - hosts.equiv file 2-7
  - UNIX client 2-7
- USER clause of CONNECT
  - statement 2-7

- User informix
  - and the message server 1-12
  - running onshowaudit 1-17
- User-defined routines
  - indexing 4-37
- USING clause
  - CREATE INDEX statement 4-52
- Utilities
  - DB-Access 3-42
  - dbload 3-34, 3-42
  - dbschema 3-55, 3-62
  - dbschema -hd 4-33
  - oncheck
    - and index sizing 3-18
    - and sbospace sizing 3-27
    - monitoring table growth 3-38
  - pe option 3-41
  - pr option 3-9, 3-25, 3-40
  - pt option 3-9
  - onload and onunload 3-4, 3-45
  - onmode
    - MQDS options 4-69
  - onstat
    - c option 3-27
    - g mgm option 4-66
    - g ppf option 3-64
    - t option 3-7

---

## V

- VARCHAR data type 3-11, 3-12, 3-47, 4-22
- Variable-length rows 3-11
- Virtual processor
  - as Windows NT thread 1-3
  - description of 1-3
  - user-defined class 1-5
- Virtual processors
  - OPT (optical disk) A-4
  - SOC, network A-4
- VPCLASS parameter
  - mentioned 1-5

---

## W

- Warm restore
  - and critical dbspaces 5-55
  - description of 5-46
  - in mixed restore 5-50
  - part of a mixed restore 5-47
  - performing a 5-54
  - steps to perform 5-53
- WHERE clause 4-30, 4-32, 4-33, 4-37, 4-46
- Windows Internet Name Service 2-6
- Windows NT
  - database server implementation
    - for A-1
  - event logs 1-11, 1-13
  - Services application 1-8
  - threads A-4
- Windows NT threads and database server threads 1-3
- Windows Sockets 2-5
- win.ini file 1-9

---

## Symbols

- \$INFORMIXDIR variable A-3
- .netrc file
  - supplying user name and password 2-7
- /etc directory (UNIX) 1-9

