Administrator's Guide

Informix Red Brick Decision Server

Version 6.1
October 2000
Part No. 000-8053

Published by Informix® Press

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

© 2000 Informix Corporation. All rights reserved. The following are trademarks of Informix Corporation or its affiliates, one or more of which may be registered in the United States or other jurisdictions:

Answers OnLine™; Ardent™; Ardent™ DataStage®;Axielle™; C-ISAM®; ClickPack™; Client SDK™; Cloudconnector™; Cloudscape™; Cloudsync™; Cloudview™; DataBlade®; Data Director™; Data Mine™; Data Mine Builder™; DataStage®; Decision FastStart™; Decision for Telecommunications Campaign Management™; Decision Frontier™; Decision Solution Suite™; Decisionscape™; Dialogue™; Dynamic Connect™; Dynamic Scalable Architecture™; Dynamic Server™; Dynamic Server.2000™; Dynamic Server™, Developer Edition™; Dynamic Server™ with Advanced Decision Support Option™; Dynamic Server™ with Extended Parallel Option™; Dynamic Server™ with MetaCube® ROLAP Option; Dynamic Server™ with Universal Data Option™; Dynamic Server™ with Web Integration Option™; Dynamic Server™, Workgroup Edition™; Dynamic Virtual Machine™; e-Iterations™; Encrypt.CSM™; Enterprise Decision Server™; E-Stage™; Extract PACK™; Formation™; Formation Architect™; Formation Flow Engine™; Foundation.2000™; Frameworks for Business Intelligence™; Frameworks Technology™; Gold Mine Data Access®; i.Decide™; i.Decide™ Web Success™; i.Financial Services™; i.Foundation™; i.Intelligence™; i.Reach™; i.Sell™; Illustra®; Informix®; Informix® COM Adapter™; Informix® Enterprise Command Center™; Informix® Extended Parallel Server™; Informix® Informed Decisions™; Informix® Inquire℠; Informix® Internet Foundation.2000™; InformixLink®; InformiXML™; Informix® Ready™; Informix® Red Brick® Decision Server™; Informix Session Proxy™; Informix® Vista™; InfoShelf™; Installation Assistant™; Interforum™; I-Spy™; Iterations®; J/Foundation™; Load PACK™; LUCID™; MaxConnect™; Media360™; Mediazation™; MetaArchitect™; MetaBroker™; MetaCube®; MetaHub™; MetaStage™; NewEra®; O2 & Design®; O2 Technology & Design®; Object Translator™; Office Connect™; ON-Bar™; OnLine Dynamic Server™; OnLine/Secure Dynamic Server™; OpenCase®; Orca™; PaVER™; Prism®; Prism & Design™; RedBack®; RedBean™; RedBeans & Design™; Red Brick® and Design; Red Brick® Data Mine™; Red Brick Decision Server™; Red Brick® Mine Builder™; Red Brick® Decisionscape™; Red Brick® Ready™; Red Brick Systems®; Regency Support®; Rely on Red Brick℠; RISQL®; Server Administrator™; Solution Design℠; STARindex™; STARjoin™; SuperTerm®; SuperView®; SureStart™; SystemBuilder™; TARGETindex™; TARGETjoin™; The Data Warehouse Company®; UniData®; UniData & Design®; Universal Data Warehouse Blueprint™; Universal Database Components™; Universal Web Connect™; UniVerse®; ViewPoint®; Virtual Table Interface™; Visionary™; Vista™; Web Integration Suite™; XML DataPort™; XML Pack™; Zero Defect Data®. The Informix logo is registered with the United States Patent and Trademark Office. The DataBlade logo is registered with the United States Patent and Trademark Office.

Documentation Team: Twila Booth, Diana Chase, Kathy Eckardt, Inge Halilovic, Ellie Lief

GOVERNMENT LICENSE RIGHTS

Software and documentation acquired by or for the US Government are provided with rights as follows: (1) if for civilian agency use, with rights as restricted by vendor's standard license, as prescribed in FAR 12.212; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by a negotiated vendor license, as prescribed in DFARS 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce this legend.

Table of Contents

Introduction

In This Introduction 3
About This Guide 3
- Types of Users 3
- Software Dependencies 4

Documentation Conventions 4
- Typographical Conventions 5
- Syntax Notation 6
- Syntax Diagrams 7
- Keywords and Punctuation 9
- Identifiers and Names 9
- Icon Conventions 10

Customer Support 11
- New Cases 12
- Existing Cases 12
- Troubleshooting Tips 13

Related Documentation 14
Additional Documentation 16
- Online Manuals 16
- Printed Manuals 16
- Online Help 16

Informix Welcomes Your Comments 17

Chapter 1 Overview of Red Brick Decision Server

In This Chapter 1-3
Database Server Technology 1-4
Database Server Components 1-5
- Red Brick Decision Server 1-6
- Parallel Table Management Utility 1-7
- RISQL Entry Tool and RISQL Reporter 1-7

Administrator Tool 1-8
Client Connector Pack 1-8
Database Server 1-9
Interprocess Communication 1-9
Warehouse API Process 1-11
Database Server Processes 1-11
Administration Daemon Process 1-12
Log Daemon Process 1-12
Process Checker Daemon 1-12
Vacuum Cleaner Daemon 1-13
Listener Thread 1-13
CTRL-C Coordination Thread 1-14
Shared Memory 1-14
Database Administration Overview 1-14
Installing Red Brick Decision Server 1-15
Planning the Database Design 1-15
Implementing the Database 1-16
Providing User Access 1-17
Loading and Unloading Data 1-18
Maintaining the Database and Tuning for Performance . . . 1-19
Planning Backup and Restore Procedures 1-19
Aroma Sample Database 1-20
Database Limits 1-20

Chapter 2 Key Concepts

In This Chapter 2-3
Data Loading 2-3
Parallel Processing 2-4
Physical Implementation of Databases 2-5
Indexes and Retrieval Strategies 2-6
Segmented Storage 2-7
Precomputed Views for Increased Query Performance 2-14
Query Sampling for Application Input 2-15
Database Directories and Files 2-15
Logical Database Names 2-16
Segment Names 2-21
Configuration and Initialization 2-21
Configuration File 2-21
Initialization Files 2-22
SET Commands 2-24

Environment Variables 2-25
Administrator Tool 2-25
Server Locale 2-27
Components of a Locale 2-27
Defining the Server Locale 2-31
Overriding the Server Locale 2-32
Ensuring Client/Server Compatibility 2-35
File Ownership and Permissions 2-37
Database Authorizations and Privileges 2-37
Versioned Databases 2-39
Referential Integrity 2-40
Load and Insert Operations 2-40
Delete Operations and Cascaded Deletes 2-41

Chapter 3 Schema Design

In This Chapter 3-3
Transaction Processing Versus Decision Support 3-3
Transaction-Processing Databases 3-4
Decision-Support Databases 3-5
Star Schemas 3-6
Performance of Star Schemas 3-8
Terminology 3-8
Simple Star Schemas 3-8
Multistar Schemas 3-14
Views 3-17
Considerations for Schema Design 3-18
Schema Building Blocks 3-20
Example: Salad Dressing Database 3-23
Analyzing Your Schema 3-24
Browsing the Dimension Tables 3-24
Querying the Fact Table 3-25
Determining Which Attributes to Include 3-25
Schema Examples 3-27
Reservation System Database 3-27
Investment Database 3-29
Health Insurance Database 3-31

Chapter 4 Planning a Database Implementation

In This Chapter 4-3
Organizing Data into Databases 4-3
Determining When to Create Additional Indexes 4-4
STAR Indexes 4-6
B-TREE Indexes 4-9
TARGET Indexes 4-10
No Indexes 4-12
Planning for TARGETjoin Processing 4-12
STARjoin Versus TARGETjoin 4-13
Administration Considerations for TARGETjoin Processing 4-14
TARGET Index DOMAIN Clause 4-18
Planning Disk Storage Organization 4-20
Estimating the Size of User Tables 4-21
Estimating the Size of Indexes 4-22
Example: Calculating Table, Index, and System Table Sizes 4-27
Estimating the Size of System Tables 4-32
Total Space for User Tables, Indexes, and System Tables . . . 4-34
Estimating Temporary Space Requirements 4-35
How Optimized Index-Building Operations Use Temporary Space 4-36
Estimating Temporary Space Values for Index-Building Operations 4-37
Temporary Space Requirements for TARGET Indexes 4-42
How Query Operations Use Temporary Space. 4-43
Estimating a QUERY_MEMORY_LIMIT Value for Queries 4-43
Estimating a MAXSPILLSIZE Value for Queries 4-44
Planning for Segmented Storage 4-45
Determining When to Use Default and Named Segments . . . 4-46
Considerations for Growing Tables 4-48
Effect of Table Growth on STAR Indexes. 4-49

Chapter 5 Creating a Database

In This Chapter 5-3
Overview 5-3
Creating the Database Structure 5-4
Database Creation Syntax 5-4
Initializing the Database 5-6
Defining a Logical Database Name 5-8
Changing the DBA Account Password 5-9
Creating the Database Objects 5-11
Creating Segments 5-12
Creating Tables 5-13
Setting the MAXSEGMENTS and MAXROWS PER SEGMENT Options 5-14
Naming Constraints for Primary and Foreign Keys 5-15
Maintaining Referential Integrity with ON DELETE 5-15
Creating Indexes 5-16
INDEX TEMPSPACE Parameters 5-17
Parallel Indexes 5-17
Loading Tables with Indexes 5-18
STAR Indexes 5-19
TARGET Indexes 5-19
Creating Views 5-19
Creating and Managing Macros 5-21
Guidelines for Macro Definitions 5-22
Availability and Scope 5-23

Chapter 6 Working with a Versioned Database

In This Chapter 6-3
Determining Whether You Need Versioning 6-4
Load Window 6-4
Increased Recoverability 6-4
Load with Periodic Commit 6-5
Providing Load Recoverability 6-6
Versioning with Vista 6-6
Dimension Table Cleaning 6-7
Costs of the Version Log 6-7
Loading Data into Versioned Databases 6-8
Understanding the Version Log 6-10
Structure of the Version Log 6-11
Versioned DELETE Operations 6-13

Understanding Frozen Versions 6-13
Loading Data with the High Recoverability Option 6-16
Controlling Versioning 6-16
Creating the Version Log 6-18
Cleaning the Version Log 6-19
Dropping the Version Log and Adding Space 6-20
Controlling Frozen Versions 6-21
Maintaining a Versioned Database 6-23
Monitoring the Version Log 6-23
Backup and Recovery 6-24
Controlling the Vacuum Cleaner 6-25
Example: Creating a Versioned Aroma Database 6-26

Chapter 7 Providing Database Access and Security

In This Chapter 7-3
Adding Database Users 7-3
Creating Operating-System Accounts for Users 7-4
Granting Database Access. 7-5
Changing Passwords 7-6
Granting Access with System Roles 7-7
DBA, RESOURCE, and CONNECT Capabilities 7-7
Granting and Revoking the DBA and RESOURCE System Roles 7-8
Granting Database Object Privileges 7-9
Granting Access with Role-Based Security 7-10
Task Authorizations. 7-12
Role Capabilities 7-14
Creating Roles. 7-15
Granting Task Authorizations 7-16
Granting Object Privileges to Roles 7-16
Granting Roles 7-18
Revoking Task Authorizations, Object Privileges, and Roles 7-21
Tracking Role Authorizations and Members 7-23
Administering Password Security 7-26
Enforcing Password Changes 7-28
Warning Users of Password Expiration 7-30
Limiting Reuse of Previous Passwords 7-31
Limiting Frequency of Password Changes 7-32

Enforcing Password Complexity and Length 7-33
Locking User Accounts After Failed Connection Attempts 7-36

Chapter 8 Managing Database Activity in an Enterprise

In This Chapter 8-3
Task Authorizations for Managing Database Activity 8-4
Administration Database 8-4
Monitoring Database Activity with Dynamic Statistic Tables . . 8-6
Read and Write Statistics 8-7
Controlling Database Activity 8-9
Bringing a Database to a Quiescent State 8-10
Activating a Database 8-10
Resetting Accumulated Statistics 8-11
Canceling a User Command 8-11
Closing a User Session 8-11
Changing User Priorities for the Current Session 8-12
Administration Daemon Process 8-12
Statistics Collection Interval 8-14
DST Refresh Interval 8-14
Event Logging 8-15
Logging Subsystem 8-16
Event Log Messages 8-21
Log Files 8-24
Configuring the Logging Subsystem 8-25
Query Logging 8-29
Controlling Advisor Logging 8-29
Advisor Log Files 8-29
What the Advisor Logs 8-30
Starting and Stopping the Advisor Log 8-31
ADMIN ADVISOR_LOG_DIRECTORY 8-32
ADMIN ADVISOR_LOG_MAXSIZE 8-33
Accounting 8-34
Accounting Process 8-35
Format of Accounting Records 8-35
Accounting Files 8-37
Configuring Accounting 8-38
Controlling Accounting 8-40

Chapter 9 Maintaining a Data Warehouse

In This Chapter 9-5
Locking Tables and Databases 9-6
Manual Table or Database Locks 9-6
Schema Locking 9-7
Types of Table Locks 9-8
Locking and Segments 9-10
Determining When to Lock a Table or Database 9-10
Specifying Wait Behavior for Database Server and TMU Locks 9-11
Setting Isolation Level for Versioned Transactions 9-12
Obtaining Information on Tables and Indexes 9-14
Checking and Repairing Table Structure 9-14
CHECK TABLE Segment Statistics 9-14
Monitoring Growth of Tables and Indexes 9-18
STAR Indexes 9-19
MAXSIZE Column 9-20
USED Column. 9-21
TOTALFREE Column 9-22
Pseudocolumns 9-23
Freespace Field 9-24
Adding Space to a Segment 9-26
Altering Segments 9-29
Using ALTER SEGMENT 9-29
Ensuring No Users Are Active 9-30
Attaching and Detaching Segments 9-31
Moving Entire Segments 9-31
Specifying a Segmenting Column 9-32
Specifying a Range 9-32
Taking a Segment Offline or Online 9-32
Clearing a Segment 9-33
Renaming a Segment 9-33
Changing PSU Sizes. 9-33
Changing PSU Location 9-34
Verifying a Segment. 9-34
Forcing a Segment into an Intact State 9-35
Recovering a Damaged Segment 9-35

Managing Optical Storage 9-37
Assigning Optical Storage 9-38
Specifying Access Behavior for Optical Segments 9-39
Specifying Index Selection with Optical Segments 9-40
Altering Tables 9-40
Adding and Dropping Columns 9-41
Changing a Column Name 9-42
Changing the Default Value for a Column 9-42
Changing the MAXSEGMENTS and MAXROWS PER SEGMENTS Values 9-42
Changing the Way Referential Integrity Is Maintained . . . 9-43
Changing the Data Type for a Column 9-43
Adding and Dropping Foreign Keys 9-44
Changing the Fill Factor for a VARCHAR Column. 9-45
Changing the Start and Step Values of a SERIAL Column 9-45
Recovering from an Interrupted ALTER TABLE Operation 9-46
Copying or Moving a Database 9-48
Full Versus Relative Pathnames 9-48
Copy7ing a Database That Contains Only Relative Pathnames 9-50
Copying a Database That Contains Full Pathnames 9-50
Moving a Database That Contains Only Relative Pathnames 9-51
Moving a Database That Contains Full Pathnames. 9-52
Modifying the Configuration File 9-52
Monitoring and Controlling a Database Server 9-59
Monitoring and Controlling a Database Server on UNIX . . 9-59
Monitoring and Controlling a Database Server on Windows. 9-62
Determining Version Information 9-64
Deleting Database Objects and Databases 9-64
Dropping Database Objects. 9-65
Deleting a Database 9-68

Chapter 10 Tuning a Warehouse for Performance

In This Chapter 10-5
Specifying Parameters with rbw.config File Entries or SET Commands 10-6
Setting Temporary Space Parameters 10-7
Temporary Space Parameters 10-8
How Temporary Space Is Allocated 10-9
TEMPSPACE 10-12
Determining Current Values 10-16
Removing Temporary Files 10-17
Setting QUERY_MEMORY_LIMIT. 10-17
Setting the Result Buffer for Long-Running Queries 10-19
RESULT BUFFER Parameter 10-20
RESULT BUFFER FULL ACTION Parameter 10-21
Setting Segment and Partial Availability Behavior 10-22
Location of Default Segments 10-22
Segment Drop Behavior 10-24
Query Behavior on Partially Available Tables 10-25
Use of Partially Available Indexes 10-28
Setting the VARCHAR Column Fill Factor 10-29
How the Database Server Uses the VARCHAR Fill Factor. . . 10-29
Effect of Fill Factor on Performance 10-30
Monitoring Accuracy of the VARCHAR Fill Factor 10-35
Modifying the VARCHAR Fill Factor 10-37
Setting the Index Fill Factor 10-38
Finding the Fill Factor Used for a Specific Index 10-41
Deciding Whether to Change Default Fill Factors. 10-41
Changing an Index Fill Factor 10-42
Creating Additional Indexes 10-43
Understanding Query Processing 10-44
Join Algorithms 10-44
Operator Model 10-47
EXPLAIN Statement 10-57
TARGETjoin Query Processing 10-61
How to Use TARGETjoin Processing 10-61
When to Use TARGETjoin Processing. 10-64
Examples 10-66
Reading EXPLAIN Output for a TARGETjoin Query 10-69
Summary and Recommendations 10-74

Estimates of Fact Table Selectivity 10-77
Example: Selectivity Estimates for Preloaded Dimensions . . 10-79
Using Synonyms to Control Fact-to-Fact Joins 10-81
Making SQL-Based Improvements 10-84
UNION Versus Interdimensional ORs 10-84
Subquery in the FROM Clause Versus Correlated Subquery 10-85

Chapter 11 Tuning a Warehouse for Parallel Query Processing

In This Chapter 11-3
Parallel Query Tuning Parameters 11-4
Enabling Parallel Query Processing 11-5
Limiting I/O Contention with the FILE_GROUP Parameter . . . 11-6
Allowing Parallelism Within Disk Groups with the GROUP Parameter 11-8
Limiting Available Tasks 11-10
TOTALQUERYPROCS 11-11
QUERYPROCS 11-11
Setting Minimum Row Requirements with ROWS_PER_TASK Parameters 11-13
ROWS_PER_SCAN_TASK 11-14
ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK 11-17
Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters 11-23
FORCE_SCAN_TASKS 11-26
FORCE_FETCH_TASKS and FORCE_JOIN_TASKS 11-27
FORCE_HASHJOIN_TASKS 11-29
Enabling Partitioned Parallelism for Aggregation 11-31
System Considerations for Parallel Tasks 11-32
Analysis of System Resources and Workload 11-34
Disk Usage 11-34
Memory Usage 11-36
CPU Allocation 11-37
Tuning for Specific Query Types 11-38
Parallel STARjoin Queries 11-38
Parallel Table Scans 11-41
SuperScan Technology 11-42
About Reasonable Values 11-42
Basic Guidelines 11-43

Appendix A **Example: Building a Database**

Appendix B **Configuration File**

Appendix C **System Tables and Dynamic Statistic Tables**

Appendix D **Example: Using Segments with Time-Cyclic Data**

Index

Introduction

About This Guide 3
Types of Users 3
Software Dependencies 4

Documentation Conventions 4
Typographical Conventions 5
Syntax Notation 6
Syntax Diagrams 7
Keywords and Punctuation 9
Identifiers and Names 9
Icon Conventions 10
Comment Icons 10
Platform Icons 10

Customer Support 11
New Cases 12
Existing Cases 12
Troubleshooting Tips 13

Related Documentation 14

Additional Documentation 16
Online Manuals 16
Printed Manuals 16
Online Help 16

Informix Welcomes Your Comments 17

In This Introduction

This introduction provides an overview of the information in this document and describes the conventions it uses.

About This Guide

This guide contains important information on administering Informix Red Brick Decision Server and provides information about how to install, configure, and use the database server. It describes features, database server concepts, and procedures for performing database server management and performance tuning tasks.

Types of Users

This guide is written for the following users:

- Database administrators
- Database server administrators
- Database architects
- Database designers
- Database developers
- Backup operators
- Performance engineers

This guide assumes that you have the following background:

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

Software Dependencies

This guide assumes that you are using Informix Red Brick Decision Server, Version 6.1, as your database server.

Red Brick Decision Server includes the **Aroma** database, which contains sales data about a fictitious coffee and tea company. The database tracks daily retail sales in stores owned by the Aroma Coffee and Tea Company. The dimensional model for this database consists of a fact table and its dimensions.

For information about how to create and populate the demonstration database, see Appendix A, "Example: Building a Database." For a description of the database and its contents, see the *SQL Self-Study Guide*.

The scripts that you use to install the demonstration database reside in ***redbrick_dir*/sample_input** on UNIX or ***redbrick_dir*\SAMPLE_INPUT** on Windows NT, where ***redbrick_dir*** is the Red Brick Decision Server directory on your system.

Documentation Conventions

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

The following conventions are discussed:

- Typographical conventions
- Syntax notation

- Syntax diagrams
- Keywords and punctuation
- Identifiers and names
- Icon conventions

Typographical Conventions

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

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



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

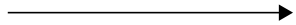
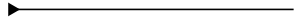
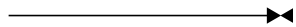
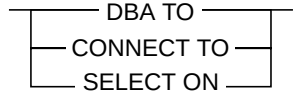
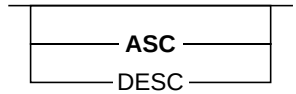
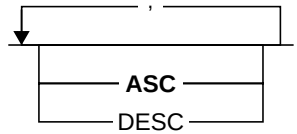
Syntax Notation

This guide uses the following conventions to describe the syntax of operating-system commands.

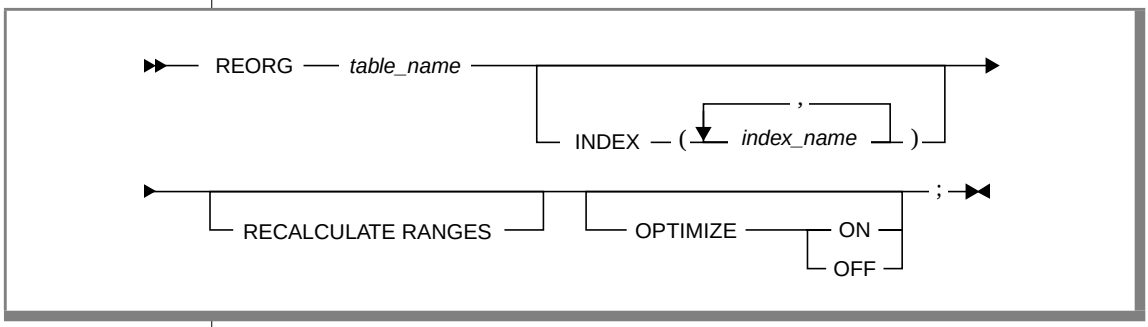
Command Element	Example	Convention
Values and parameters	*table_name*	Items that you replace with an appropriate name, value, or expression are in *italic* type style.
Optional items	[ ]	Optional items are enclosed by square brackets. Do not type the brackets.
Choices	ONE | TWO	Choices are separated by vertical lines; choose one if desired.
Required choices	{ONE | TWO}	Required choices are enclosed in braces; choose one. Do not type the braces.
Default values	ONE | TWO	Default values are underlined, except in graphics where they are in **bold** type style.
Repeating items	name, …	Items that can be repeated are followed by a comma and an ellipsis. Separate the items with commas.
Language elements	( ) , ; .	Parentheses, commas, semicolons, and periods are language elements. Use them exactly as shown.

Syntax Diagrams

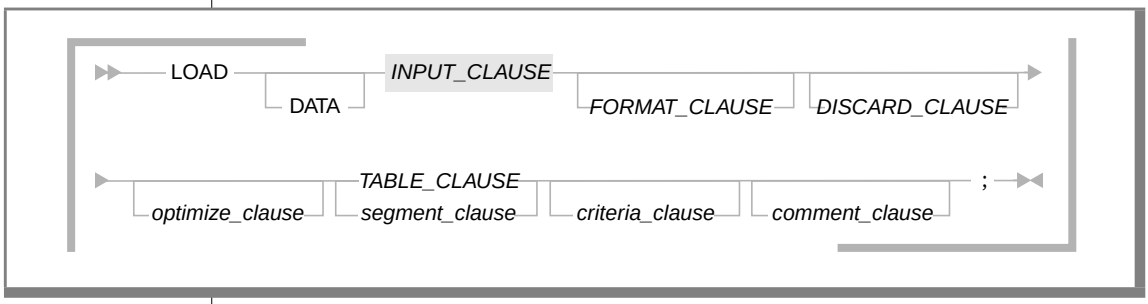
This guide uses diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands.

Component	Meaning
	Statement begins.
	Statement syntax continues on next line. Syntax elements other than complete statements end with this symbol.
	Statement continues from previous line. Syntax elements other than complete statements begin with this symbol.
	Statement ends.
	Required item in statement.
	Optional item.
	Required item with choice. One and only one item must be present.
	Optional item with choice. If a default value exists, it is printed in **bold**.
	Optional items. Several items are allowed; a comma must precede each repetition.

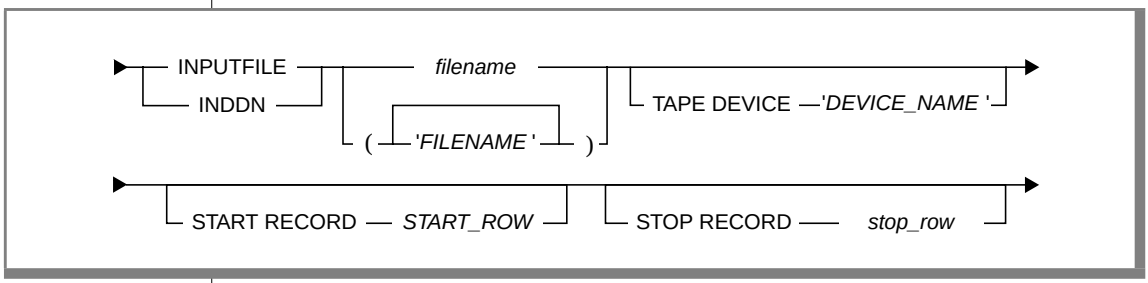
The preceding syntax elements are combined to form a diagram as follows.



Complex syntax diagrams such as the one for the following statement are repeated as point-of-reference aids for the detailed diagrams of their components. Point-of-reference diagrams are indicated by their shadowed corners, gray lines, and reduced size.



The point-of-reference diagram is then followed by an expanded diagram of the shaded portion—in this case, the *INPUT_CLAUSE*.



Keywords and Punctuation

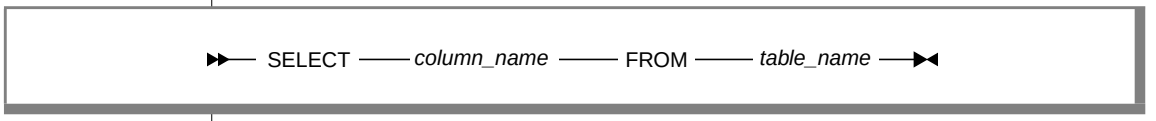
Keywords are words reserved for statements and all commands except system-level commands. When a keyword appears in a syntax diagram, it is shown in uppercase characters. You can write a keyword in uppercase or lowercase characters, but you must spell the keyword exactly as it appears in the syntax diagram.

Any punctuation that occurs in a syntax diagram must also be included in your statements and commands exactly as shown in the diagram.

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples. You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

Icon Conventions

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

Comment Icons

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

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

Platform Icons

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

Icon	Description
	Identifies information that is specific to the UNIX and Linux operating systems
	Identifies information that is specific to Windows NT, Windows 95, and Windows 98 environments
	Identifies information that is specific to the Windows NT and Windows 2000 environments

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

Customer Support

Please review the following information before contacting Informix Customer Support.

If you have technical questions about Red Brick Decision Server but cannot find the answer in the appropriate document, contact Informix Customer Support as follows:

Telephone	1-800-274-8184 or 1-913-492-2086 (7 A.M. to 7 P.M. Central Time, Monday through Friday)
Internet access	http://www.informix.com/techinfo

For nontechnical questions about Red Brick Decision Server, contact Informix Customer Support as follows:

Telephone	1-800-274-8184 (7 A.M. to 7 P.M. Central Time, Monday through Friday)
Internet access	http://www.informix.com/services

New Cases

To log a new case, you must provide the following information:

- Red Brick Decision Server version (Refer to "Determining Version Information" on page 9-64.)
- Platform and operating-system version
- Error messages returned by Red Brick Decision Server or the operating system
- Concise description of the problem, including any commands or operations performed before you received the error message
- List of Red Brick Decision Server or operating-system configuration changes made before you received the error message

For problems concerning client-server connectivity, you must provide the following additional information:

- Name and version of the client tool that you are using
- Version of Informix Red Brick ODBC Driver or Informix Red Brick JDBC Driver that you are using, if applicable
- Name and version of client network or TCP/IP stack in use
- Error messages returned by the client application
- Server and client locale specifications

Existing Cases

The support engineer who logs your case or first contacts you will always give you a case number. This number is used to keep track of all the activities performed during the resolution of each problem. To inquire about the status of an existing case, you must provide your case number.

Troubleshooting Tips

You can often reduce the time it takes to close your case by providing the smallest possible reproducible example of your problem. The more you can isolate the cause of the problem, the more quickly the support engineer can help you resolve it:

- For SQL query problems, try to remove columns or functions or to restate WHERE, ORDER BY, or GROUP BY clauses until you can isolate the part of the statement causing the problem.
- For Table Management Utility load problems, verify the data type mapping between the source file and the target table to ensure compatibility. Try to load a small test set of data to determine whether the problem concerns volume or data format.
- For connectivity problems, issue the **ping** command from the client to the host to verify that the network is up and running. If possible, try another client tool to see if the same problem arises.

Related Documentation

The standard documentation set for Red Brick Decision Server includes the following documents.

Document	Description
This guide	Describes warehouse architecture, supported schemas, and other concepts relevant to databases. Procedural information for designing and implementing a database, maintaining a database, and tuning a database for performance. Includes a description of the system tables and the configuration file.
Client Installation and Connectivity Guide	Includes procedures for installing Red Brick ODBC Driver, Red Brick JDBC Driver, RISQL Entry Tool, and RISQL Reporter on client systems. Describes how to access Red Brick Decision Server using ODBC for C and C++ applications and JDBC for Java applications.
Informix Vista User's Guide	Describes the Informix Vista aggregate computation and management system. Illustrates how Vista improves query performance by automatically rewriting queries to use aggregate tables; describes how the Advisor recommends the best set of aggregate tables based on data collected daily; and explains how aggregate tables are maintained when their detail tables are updated.
Installation and Configuration Guide	Provides installation and configuration information, as well as platform-specific material, about Red Brick Decision Server and related products. Customized for either UNIX and Linux and or Windows NT and Windows 2000.
Messages and Codes Reference Guide	Contains a complete listing of all informational, warning, and error messages generated by Informix Red Brick Decision Server products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.

(1 of 2)

Document	Description
The release notes	Contains information pertinent to the current release that was unavailable when the documents were printed.
SQL Reference Guide	Is a complete language reference for the Informix Red Brick SQL implementation and RISQL extensions for Red Brick Decision Server databases.
SQL Self-Study Guide	Provides an example-based review of SQL and introduction to the RISQL extensions, the macro facility, and **Aroma**, the sample database.
Table Management Utility Reference Guide	Describes the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the **rb_cm** copy management utility.

(2 of 2)

Additional references you might find helpful include:

- An introductory-level book on SQL
- An introductory-level book on relational databases
- Documentation for your hardware platform and operating system

Additional Documentation

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

- Online manuals
- Printed manuals
- Online help

Online Manuals

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

Printed Manuals

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

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

Online Help

Informix provides online help with each graphical user interface (GUI) that displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the online help.

Informix Welcomes Your Comments

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

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

Send electronic mail to us at the following address:

doc@informix.com

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

We appreciate your suggestions.

Chapter

1

Overview of Red Brick Decision Server

Database Server Technology 1-4

Database Server Components 1-5
- Red Brick Decision Server 1-6
- Parallel Table Management Utility 1-7
- RISQL Entry Tool and RISQL Reporter 1-7
- Administrator Tool 1-8
- Client Connector Pack 1-8

Database Server 1-9
- Interprocess Communication 1-9
- Warehouse API Process 1-11
- Database Server Processes 1-11
- Administration Daemon Process 1-12
- Log Daemon Process 1-12
- Process Checker Daemon 1-12
- Vacuum Cleaner Daemon 1-13
- Listener Thread 1-13
- CTRL-C Coordination Thread 1-14
- Shared Memory 1-14

Database Administration Overview 1-14
- Installing Red Brick Decision Server 1-15
- Planning the Database Design 1-15
- Implementing the Database 1-16
- Providing User Access 1-17
  - Initialization Files 1-17
  - Macros 1-17

Loading and Unloading Data 1-18
Loading Concurrently with Queries 1-18
Exporting Query Results 1-18
Maintaining the Database and Tuning for Performance 1-19
Planning Backup and Restore Procedures 1-19

Aroma Sample Database 1-20

Database Limits 1-20

In This Chapter

Informix Red Brick Decision Server is a relational database management system (RDBMS) designed for data warehouse, data mart, and online analytical processing (OLAP) applications. Red Brick Decision Server delivers high performance for query processing and data loading, and ease of administration. It provides a rich set of specialized features for applications that range from a few gigabytes to tens of terabytes and from a few users to thousands of users.

Red Brick Decision Server can scale from the workgroup to the enterprise. It is built for a client/server environment using industry-standard open database connectivity (ODBC) and Java database connectivity (JDBC), and it is accessed using industry-standard SQL. The database server SQL extensions, called RISQL, simplify analyses in commonly used business calculations. The Vista, STARjoin, STARindex, TARGETjoin, and TARGETindex technologies provide unparalleled ad hoc query and analysis performance for very large databases with various schema designs. Managers and analysts can pose numerous and creative queries to quickly receive the information they need and make good business decisions with similar speed and confidence.

This chapter contains the following sections:

- Database Server Technology
- Database Server Components
- Database Server
- Database Administration Overview
- Aroma Sample Database
- Database Limits

Database Server Technology

Red Brick Decision Server is designed to provide a relational database environment well suited for the needs of analysts and business managers performing strategic data analysis. Such an environment requires the ability to accommodate very large databases, to load data quickly, to formulate meaningful queries, and to respond quickly to those queries.

Red Brick Decision Server meets these requirements through the following features:

- Dimensional segmentation, which allows the data and indexes of a table to be distributed across multiple independent physical storage units to provide partial access during data loading, improved query performance, and improved incremental backup and restore operations.
- Parallel processing on single-processor and symmetric multi-processor (SMP) systems. Parallel processing can be used in query processing, multiuser relation scans, index building, and data loading.
- Red Brick extensions to SQL, called RISQL, which include functions for rank, moving sum, moving average, cumulative total, n-tile analysis, and market share. These functions are specifically designed to take advantage of Red Brick Decision Server technology to provide answers to complex queries submitted to decision-support databases.

  For more information, refer to the *SQL Reference Guide*, the *SQL Self-Study Guide*, and the *RISQL Entry Tool and RISQL Reporter User's Guide*.
- The Parallel Table Management Utility (Parallel TMU), which loads, indexes, and performs referential integrity checking on data in a batch process, as well as performing other administrative functions. Its Auto Aggregate mode allows aggregation of new data with existing data during the load process. For optimized data loading and index building, many TMU operations run in parallel.

  For more information, refer to the *Table Management Utility Reference Guide*.

- Informix Vista aggregate computation and management, which automatically optimizes aggregate queries, maintains precomputed aggregate data when detail tables are loaded, and offers advice on the optimal aggregation strategy for the query workload. For more information, refer to the *Informix Vista User's Guide*.
- Support for time-cyclic data with date data types and the ability to segment data by date-based values.
- A state-of-the-art RDBMS that allows arbitrary join paths between tables.
- Proprietary indexing technologies, STARindex and TARGETindex, which provide fast data retrieval.
- A macro facility that simplifies creation of reusable generalized queries.

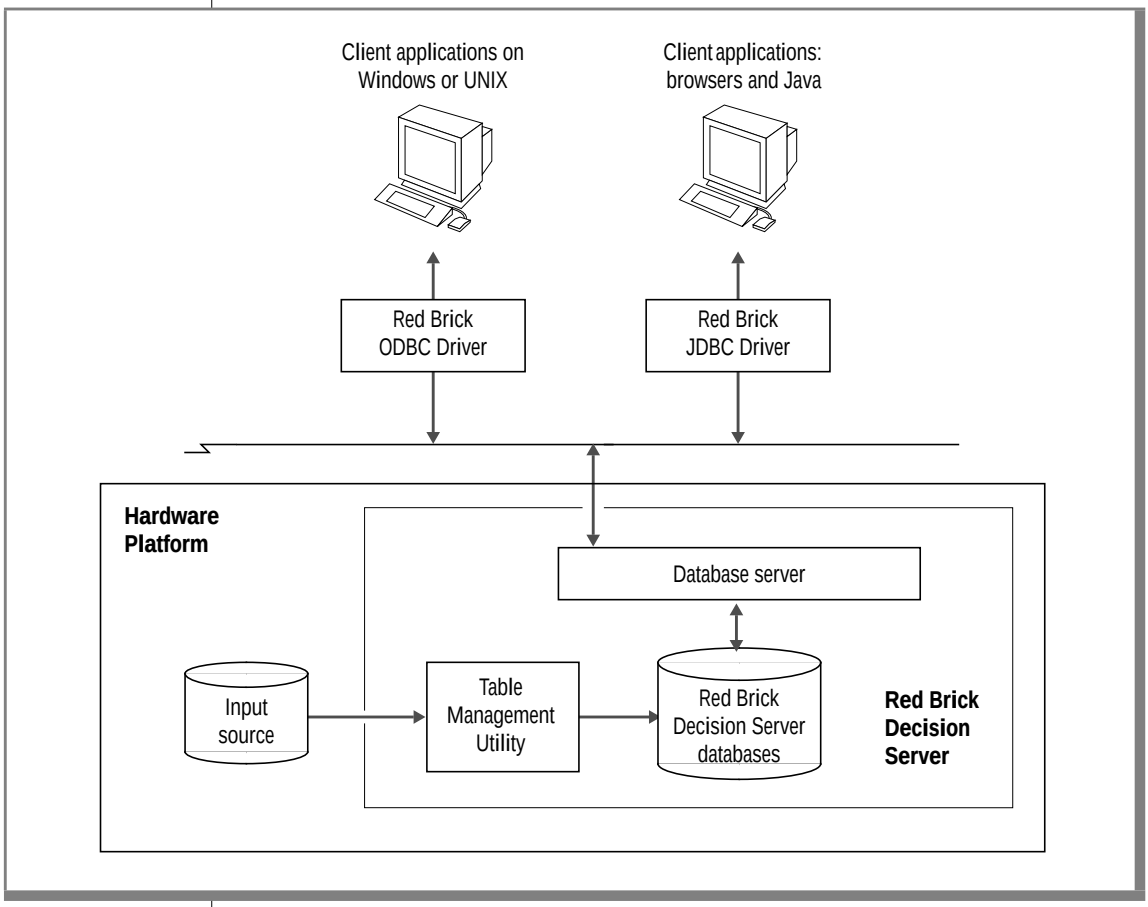
Database Server Components

The database server consists of the following software components:

- Red Brick Decision Server
- Parallel Table Management Utility (Parallel TMU)
- RISQL Entry Tool, a command-line entry tool, and RISQL Reporter, a report generator
- Administrator tool
- Informix Vista, an aggregate computation and management system. All Vista features are fully integrated into the database server, the Parallel TMU, and the Administrator tool.
- Informix Red Brick Client Connector Pack, which contains Red Brick ODBC Driver and Red Brick JDBC Driver, implementations of application programming interfaces (API)

The following figure illustrates the components of Red Brick Decision Server.

Figure 1-1
Red Brick Decision Server Components



Red Brick Decision Server

Red Brick Decision Server accepts SQL statements and delivers the results to client applications such as the RISQL Entry Tool, the RISQL Reporter, or any other client tool connected to the database server through the API. On Windows, Red Brick Decision Server runs as a Windows service. The database server functions are performed by the processes described in "Database Server" on page 1-8.

Parallel Table Management Utility

The Parallel Table Management Utility (Parallel TMU) loads data into a database. It accepts data in a variety of different formats, performs data type conversions and optional data manipulation functions, and then loads the data into the user tables. During the load process, the TMU enforces referential integrity of the data by validating the primary key / foreign key relationships.

The TMU can perform either full or incremental loads of a table and versioned loads with periodic commit intervals. The TMU can also be restarted to continue the load process after interrupts or errors.

In addition to loading data, the TMU has the following capabilities:

- Upgrade databases to run with a newer version of Red Brick Decision Server
- Reorganize indexes to improve performance as tables are modified over time
- Unload and reload data to facilitate moving a database or loading data into other tools for analysis
- Automatically generate new rows necessary for referential integrity
- Perform data aggregations as the data is loaded (Auto Aggregate mode)
- Use multiple processors for load operations (Parallel TMU)
- Use multiple processors for parallel data conversion and index building

For information about the TMU, refer to Appendix A, "Example: Building a Database," and to the *Table Management Utility Reference Guide*.

RISQL Entry Tool and RISQL Reporter

The RISQL Entry Tool is a command-line client tool that provides interactive access to Red Brick Decision Server. It is designed for use primarily by database administrators and application developers to enter RISQL queries, retrieve data, and perform other database administration functions. The RISQL Reporter tool provides all the functionality of the RISQL Entry Tool, plus report-formatting capability.

The RISQL Entry Tool and the RISQL Reporter can be used from a UNIX workstation, a network terminal emulator, or a 32-bit Windows 95 or Windows NT client connected to Red Brick Decision Server. For more information about these tools, refer to the *RISQL Entry Tool and RISQL Reporter User's Guide*.

Administrator Tool

The Administrator tool provides graphical database administration. This client tool runs on Windows-based computers and can be used to connect to databases on either UNIX or Windows to perform many database administration tasks, including segmentation.

For more information, refer to "Administrator Tool" on page 2-25.

Client Connector Pack

The Client Connector Pack contains two application programming interfaces (APIs), ODBC and Red Brick JDBC Driver.

The Red Brick ODBC Driver allows a wide variety of ODBC-compliant database applications to work with Red Brick Decision Server. This API allows you to use front-end applications, such as Microsoft Access, to access information in Red Brick Decision Server.

The Red Brick JDBC Driver allows a wide variety of JDBC-compliant database applications to work with Red Brick Decision Server. Java database connectivity (JDBC) is the JavaSoft specification of a standard API that allows Java programs to access database management systems. The Red Brick JDBC Driver consists of a set of interfaces and classes written in the Java programming language.

For further information about these APIs, refer to the *Client Installation and Connectivity Guide*.

Database Server

UNIX

Red Brick Decision Server is implemented on UNIX by a set of cooperating processes that communicate using the UNIX System V Interprocess Communication (IPC) mechanism, shared memory, and semaphores. A single process, the warehouse daemon, starts and monitors separate database server processes for each user session. This implementation is well suited for multiprocessor systems because each database server process can run on a different processor.

For more information on database server processes, refer to "Monitoring and Controlling a Database Server on UNIX" on page 9-59. ♦

WIN NT/2000

Red Brick Decision Server is implemented on Windows by a single multi-threaded process that runs as a Windows service. A Windows service contains one or more threads that are always running and can start and stop other threads. On Windows, threads run as processes. The individual threads communicate using shared memory. A single thread, the Red Brick API **(rbwapid**) thread, starts and monitors separate database server threads for each user session. The Table Management Utility (TMU) runs as a separate process that communicates with the Red Brick Decision Server service.

Red Brick Decision Server for Windows supports the following features:

- WinSock network protocol stack, Version 2.0
- Windows unified logon
- Unattended installation using Systems Management Server

For more information on database server threads, refer to "Monitoring and Controlling a Database Server on Windows" on page 9-62. ♦

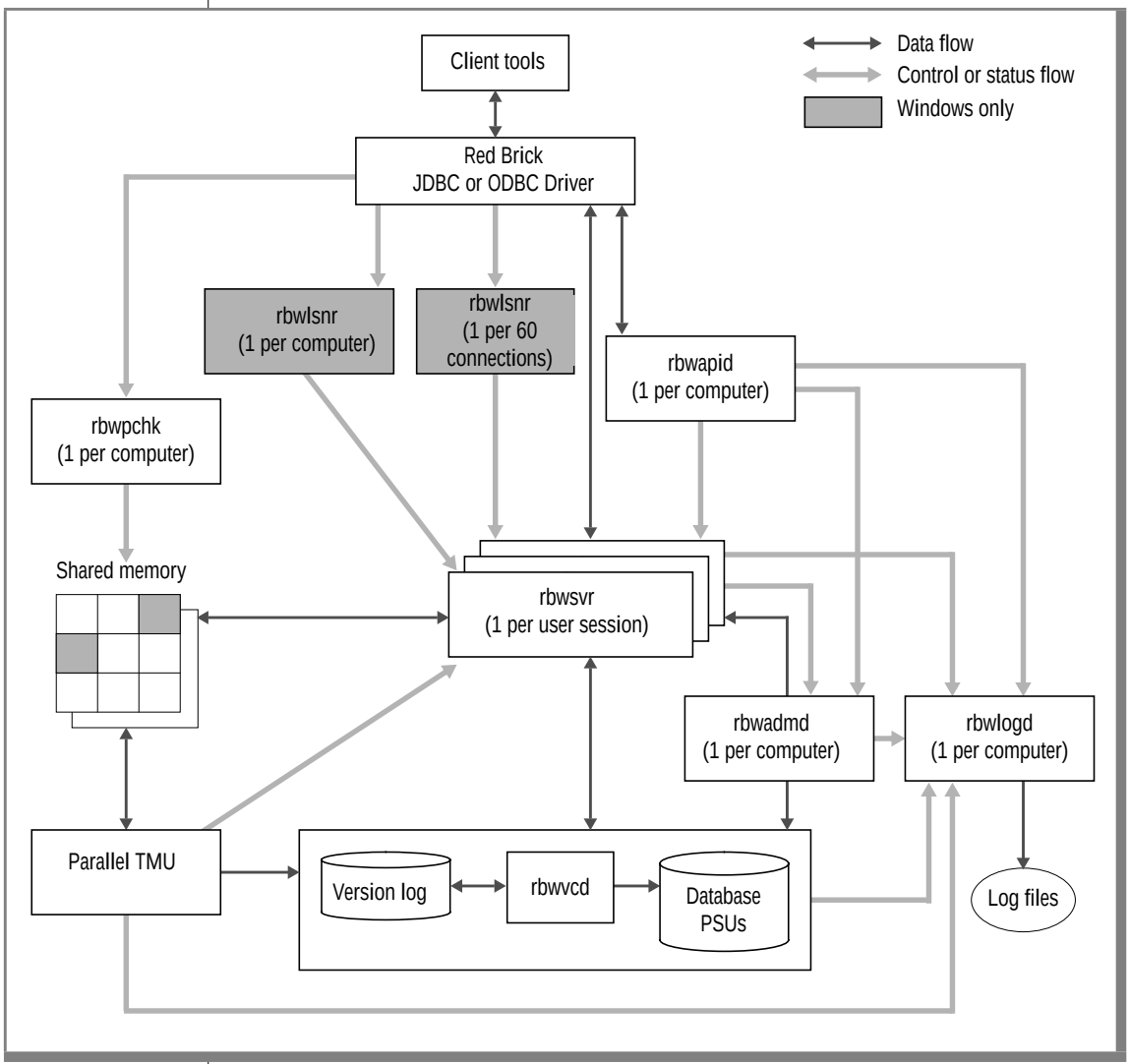
Interprocess Communication

The following figure illustrates interprocess communication among the database server processes. When a user uses a client tool to access a database, the client tool communicates with the warehouse daemon and server processes through the Red Brick ODBC Driver or Red Brick JDBC Driver. When a user accesses the database with RISQL Entry Tool or RISQL Reporter, these tools communicate directly with the warehouse daemon and server processes.

WIN NT/2000

On Windows, the Red Brick Decision Server service runs as a single multi-threaded process. The threads run as processes with names corresponding to the processes on UNIX. ♦

Figure 1-2
Interprocess Communication



Warehouse API Process

The warehouse daemon process (**rbwapid**) manages the separate database server processes (**rbwsvr**) and the communication between the separate server and associated client processes. The **rbwapid** process creates and controls the number of database server processes, manages exceptional conditions, and handles the termination and cleanup of the database server processes. In a standard configuration, the warehouse daemon process is started automatically at system startup and runs continuously.

For more information, refer to "Monitoring and Controlling a Database Server" on page 9-59.

Database Server Processes

Red Brick Decision Server uses a process-per-user architecture in which a new process named **rbwsvr** is created for each user session accessing the database. A database server process accepts SQL statements from RISQL Entry Tool, RISQL Reporter, or any other client tool connected to Red Brick Decision Server through Red Brick ODBC Driver or Red Brick JDBC Driver, checks the statement syntax, executes the statement, and returns any output to the client. Each client session is serviced by its own database server process. Each database server process exists until the client terminates the session. Additional database server processes are created as needed as children of the client-connected server process to perform parallel query processing.

For aggregate maintenance purposes, the TMU communicates directly with the database server. This communication allows database server-based changes to precomputed views to be performed as part of a single TMU transaction that updates a detail table. For information about aggregate maintenance, refer to the *Informix Vista User's Guide*.

Administration Daemon Process

The administration daemon process (**rbwadmd**) collects statistics for the dynamic statistic tables (DSTs) and performs the actions specified by ALTER SYSTEM statements. The **rbwadmd** process is started when the **rbwapid** process is started.

For more information about the **rbwadmd** process, refer to "Administration Daemon Process" on page 8-12.

Log Daemon Process

The log daemon process (**rbwlogd**) writes records to the log file when various events occur in Red Brick Decision Server. This daemon is started automatically when the **rbwapid** process starts. The database administrator can specify which events to log. If nothing is specified, the log daemon logs only a restricted set of events intended to help the Informix Customer Support Center diagnose problems.

For more information about the **rbwlogd** process, refer to "Log Daemon" on page 8-16.

Process Checker Daemon

The process checker (**rbwpchk**) is a process that looks for abnormally terminated connections and cleans up any shared resources that might be left by the abnormal termination. This ensures proper cleanup of the system even when processes are terminated outside the control of Red Brick Decision Server (for example, with a **kill -9** command on UNIX or a thread exit or End Task request from the Task Manager on Windows). The process checker is started by the **rbwapid** process.

Vacuum Cleaner Daemon

The vacuum cleaner daemon (**rbwvcd**) is present in versioned databases. There is one **rbwvcd** process per database. The purpose of the vacuum cleaner is to merge committed data from the version log into the database files and then to free the space in the version log. The vacuum cleaner is started when you enable versioning or when the first connection is made to a versioned database.

When changes occur in a versioned database, the changed blocks are initially written to the version log. While these changes are taking place, the existing committed blocks are available for query (read) operations. After the new blocks are committed to the version log but before the new blocks are moved back to the database files, query operations access the new blocks in the version log.

The vacuum cleaner daemon process waits until no users are reading the old version of the database and then proceeds to *vacuum* the changed blocks from the version log back to the database files. After the data is moved from the version log, the vacuum cleaner removes those blocks from the version log, freeing space in the version log for more versioning transactions.

For more information on the version log, refer to Chapter 6, “Working with a Versioned Database.”

WIN NT/2000

Listener Thread

The listener thread (**rbwlsnr**) is a thread in the Red Brick Decision Server service that listens for incoming connections. The listener thread reads the **rbw.config** file and determines the port on which to listen and how many simultaneous connections to allow (based on the value of the MAX_SERVERS parameter). Starting the database service automatically starts the listener thread.

WIN NT/2000

CTRL-C Coordination Thread

The CTRL-C coordination thread (**rbwconc**) listens for CTRL-C interrupts from the various clients connected to Red Brick Decision Server. This thread is started by the listener thread, and there is one CTRL-C coordination thread per 60 concurrent connections.

When a CTRL-C or cancel operation is detected from a client application, the **rbwconc** thread ensures that the **rbwsvr** thread that requested the cancel operation terminates the operation in a clean and consistent manner.

Shared Memory

All database processes access global shared memory. For versioned databases, a portion of shared memory is allocated for each database.

Database Administration Overview

Administration of Red Brick Decision Server databases includes the following activities:

- Installation, which includes preparing the database environment, installing the software, and setting up and maintaining database directories, files and administrator account
- Database design, which includes designing the schema and planning the physical storage and index strategies
- Database implementation, which includes creating the system tables, segments, user tables, and indexes and providing user access to the database
- Activities related to security and user access, such as granting and revoking access, creating macros, and providing initialization files
- Data load and unload, including versioning capabilities as needed for concurrent queries and updates

- Tuning the database to achieve the optimal performance based on equipment, database, and user requirements
- Maintenance tasks such as monitoring space and query performance, reallocating space, and rebuilding tables and indexes as needed, and performing periodic backups

The following sections provide a brief overview of these activities and references to more detailed information.

Installing Red Brick Decision Server

Red Brick Decision Server software is installed and verified from a CD-ROM device using a menu-driven script provided by Informix. The database administrator determines where to install the software, creates a directory in that location, and runs the installation script from the CD-ROM. Menu selections then drive the installation and verification process, prompting for configuration information, installing the software, creating a configuration file, and creating and loading the **Aroma** sample database. Subsequent installations of maintenance releases are also performed from the installation script.

The database administrator must create a specific administrator account to use for the installation and other administrative tasks. This account is referred to throughout Red Brick Decision Server publications as the **redbrick** user. The installation process ensures correct file access for all users.

For more information about the installation procedure, refer to the *Installation and Configuration Guide*.

Planning the Database Design

Red Brick Decision Server supports all types of database schemas, but the database designer should choose a schema that works well with the type of data warehouse that will be implemented. To implement a schema, the database administrator must first determine how to store the data. Effective planning for disk storage requires an estimate of the sizes of database tables and indexes and knowledge of which file systems can accommodate the tables and indexes. Segmented storage can be used to accommodate large databases or to improve data access and loading performance. Referential integrity and how it is maintained also affect the design of a database.

For information about referential integrity, segmented storage, and other technical concepts, refer to Chapter 2, "Key Concepts." For information about schema design, refer to Chapter 3, "Schema Design." For information about index strategies, physical storage, and size estimation, refer to Chapter 4, "Planning a Database Implementation."

Implementing the Database

Databases are created and deleted by utilities provided with the database server. These utilities are executed as the administrative user (**redbrick**).

To create a new database, use the **rb_creator** utility on UNIX or the **dbcreate** utility on Windows. Database access is automatically granted to a predefined user name and password. To ensure database security, the database administrator should immediately change this password. The administrator can then grant access to all other database users.

If user-defined segments are to be used for user tables or indexes, they are specified with CREATE SEGMENT statements. User tables and other database objects are then created with CREATE statements, which are entered with the RISQL Entry Tool, either interactively or from file input, or with other client tools that accept SQL input. Information about segments, tables, indexes, and other database objects is stored in the system tables. For a list of system tables, refer to Appendix C, "System Tables and Dynamic Statistic Tables." For information about the RISQL Entry Tool, refer to the *RISQL Entry Tool and RISQL Reporter User's Guide*.

For information about using the **rb_creator** or **dbcreate** utility, refer to "Creating the Database Structure" on page 5-4. For a complete description of the CREATE and GRANT statements, refer to the *SQL Reference Guide*.

The **rb_deleter** utility on UNIX or the **dbcreate** utility on Windows deletes database files. Execution of the utility requires write permission for the database files and for the parent directory. For information about these utilities, refer to "Deleting a Database" on page 9-68.

Providing User Access

The database administrator creates user accounts, sets object privileges, and establishes role-based security using GRANT and REVOKE statements entered with the RISQL Entry Tool, the RISQL Reporter, or other client tools that accept SQL input.

For information about user access and security, refer to Chapter 7, "Providing Database Access and Security." For information about managing your database, refer to Chapter 8, "Managing Database Activity in an Enterprise."

Initialization Files

Each time a user starts a session to access a database, that session is initialized by database server initialization files (named .**rbwrc**), which might exist on the global, database, or user levels to provide startup information for that session. Additional initialization files (named .**rbretrc**) provide setup information for the RISQL Entry Tool and RISQL Reporter.

For more information refer to "Initialization Files" on page 2-22.

Macros

Each user session is affected by macros defined for warehouse databases. The database administrator and privileged users can define the following types of macros to simplify creation of reusable generalized queries:

- A public macro available to all users of a given database
- A private macro for a given database available only to the macro creator
- A temporary macro that exists only during that session for its creator

For more information about SQL macros, refer to "Creating and Managing Macros" on page 5-21 and to the *SQL Reference Guide*.

Loading and Unloading Data

After the system and user tables have been created, use the Table Management Utility (TMU) to load data into the database and build the indexes used for data retrieval. Data can be loaded either in bulk or incrementally. Although the load performance is better when the data is ordered, the TMU can also load unordered data.

The TMU also performs automatic aggregate maintenance. When detail tables are loaded, their associated precomputed views are either incrementally updated or fully rebuilt. For information about aggregate maintenance, refer to the *Informix Vista User's Guide*.

For information about the TMU and loading or unloading tables, refer to Appendix A, "Example: Building a Database," and to the *Table Management Utility Reference Guide*.

Loading Concurrently with Queries

To perform updates concurrently with queries, use the versioning feature. For more information on versioning, see Chapter 6, "Working with a Versioned Database."

Exporting Query Results

To export query results efficiently to specified files, use the EXPORT feature. For information about the EXPORT statement, refer to the *SQL Reference Guide*. For information about task authorization for this feature, refer to "Task Authorizations" on page 7-12.

Maintaining the Database and Tuning for Performance

As a database is modified over time, the database administrator must perform the following maintenance tasks:

- Monitoring query performance

  As circumstances and the database environment change, the database administrator can improve performance by modifying configuration parameters, adjusting memory limits, providing temporary space allocations, reorganizing tables to improve data storage and access, and creating new indexes and aggregate tables as needed.
- Monitoring database storage requirements and allocating additional space as needed to accommodate growing databases
- Altering tables and segments to reflect changes in data or the database
- Performing periodic backups to prevent unrecoverable data loss
- Monitoring and controlling database activities

For information about database maintenance and performance tuning, refer to Chapter 9, "Maintaining a Data Warehouse," and Chapter 10, "Tuning a Warehouse for Performance." For more information about managing databases in an enterprise, refer to Chapter 8, "Managing Database Activity in an Enterprise."

Planning Backup and Restore Procedures

Every database administrator should have a recovery plan in case a database is damaged and becomes unusable. If your database is modified by incremental loads or insert, update, or delete operations, or if you cannot retain all of the input files used to create the database, back up the database periodically for protection in case of system or software failure. In determining how often to back up a database, you must balance the amount of data at risk and the time required for a backup.

You can use the file-oriented backup and restore facilities provided for your operating system; for example, the UNIX-based **dump** and **restore** commands or similar utility programs available for Windows.

Aroma Sample Database

Aroma, a small database for analyzing sales at a chain of specialty coffee stores, is used for many examples in this guide, the *SQL Reference Guide*, and the *SQL Self-Study Guide*. The **Aroma** database is installed during the database server installation. For information on how to create and use the **Aroma** database, see Appendix A, "Example: Building a Database."

Database Limits

The following limits apply to all Red Brick Decision Server databases, SQL, and the RISQL extensions:

- A database can contain a maximum of 32,767 tables.
- Each session can contain a maximum of 4,096 temporary tables.
- A database can contain a maximum of 61,439 segments.
- A segment can contain a maximum of 250 files, and a file can be a maximum of 2 gigabytes. Therefore, a segment can contain a maximum of 500 gigabytes.
- A table can have a maximum of 7280 columns.
- A table can have a maximum of 256 foreign keys.
- A table can contain a maximum of 2^{48} rows.
- A row in a table can contain a maximum of 8178 bytes of data if it contains fewer than 8 columns. If it contains 8 or more columns, this number is reduced by 1 byte for every 8 columns. The maximum length of a row that contains variable-length character data is 4 bytes fewer for each VARCHAR column in the table.
- Queries can contain a maximum row size of approximately 8 kilobytes in intermediate and final result tables. This limit is equivalent to the maximum size of a row in a table.
- The maximum length of the text for a macro definition is 1024 bytes.
- The maximum length of a CHAR or VARCHAR column is 1024 bytes.

- The maximum length of a string literal is 1024 bytes.
- The maximum length of a database identifier is 128 bytes.
- The maximum length of a database password is 128 characters. For the RISQL Entry Tool and RISQL Reporter, the maximum length for passwords supplied in response to the prompt is 8 characters.



Warning: *Do not use these figures to estimate database file sizes. For information on calculating space requirements for tables and indexes, refer to Chapter 4, "Planning a Database Implementation."*

Chapter

2

Key Concepts

Data Loading 2-3

Parallel Processing 2-4

Physical Implementation of Databases 2-5
- Indexes and Retrieval Strategies 2-6
- Segmented Storage 2-7
  - Named and Default Segments 2-7
  - Implementation 2-8
  - PSU Size and Growth 2-9
  - Distributing Data Among Segments 2-10
  - Online and Offline Segments 2-12
  - Partial Availability of Tables and Indexes 2-13
- Precomputed Views for Increased Query Performance 2-14

Query Sampling for Application Input 2-15

Database Directories and Files 2-15
- Logical Database Names 2-16
- Segment Names 2-21

Configuration and Initialization 2-21
- Configuration File 2-21
- Initialization Files 2-22
  - .rbwrc Files 2-22
  - .rbretrc Files 2-23
  - ODBC Initialization Files 2-24
- SET Commands 2-24
- Environment Variables 2-25
- Administrator Tool 2-25

Server Locale 2-27
Components of a Locale 2-27
Language 2-28
Territory 2-28
Code Set 2-28
Collation Sequence 2-29
Defining the Server Locale 2-31
System Table References to Locales. 2-31
Nontranslated Text 2-32
Overriding the Server Locale 2-32
Specifying a Locale for a Client Tool 2-33
Setting the RB_NLS_LOCALE Environment Variable 2-33
Ensuring Client/Server Compatibility 2-35
Code Set Conversions 2-35
Message System 2-36

File Ownership and Permissions 2-37

Database Authorizations and Privileges. 2-37

Versioned Databases 2-39

Referential Integrity 2-40
Load and Insert Operations. 2-40
Delete Operations and Cascaded Deletes 2-41

In This Chapter

To perform database administration functions effectively, you should understand the key concepts of Red Brick Decision Server. This chapter consists of the following sections:

- Data Loading
- Parallel Processing
- Physical Implementation of Databases
- Query Sampling for Application Input
- Database Directories and Files
- Configuration and Initialization
- Server Locale
- File Ownership and Permissions
- Database Authorizations and Privileges
- Versioned Databases
- Referential Integrity

Data Loading

Data is loaded in a bulk process using the Parallel Table Management Utility (Parallel TMU), which indexes data, verifies referential integrity, and optionally maintains aggregate tables as the load progresses. Data loading is CPU intensive, so more powerful CPUs improve load times roughly in proportion to their CPU speed rating. Despite the CPU-intensive nature of the load process, loading a Red Brick Decision Server database is as fast as or faster than loading databases for other RDBMSs.

The TMU supports the following types of data:

- Single-byte and multibyte character data, including ASCII and IBM U.S. EBCDIC character data
- Integer and numeric data for each supported hardware platform and for IBM System / 370
- IBM System / 370 packed-decimal and zoned-decimal data

Data can be loaded from:

- Disk files
- A pipe from another system program
- TAR format or ANSI standard label tapes ♦

UNIX

For tables that have multiple user-defined indexes, parallel index creation reduces the total time required to create the indexes and is often more convenient than creating each index separately.

User-defined indexes can be defined either before or after data is loaded. For more information on loading tables with indexes, refer to "Creating Indexes" on page 5-16.

Parallel Processing

Red Brick Decision Server runs on a wide range of hardware systems, from single microprocessor systems to large multiprocessor systems. The administrator can specify the degree to which database server and TMU processes take advantage of multiple processors, balancing the system load against the performance requirements for loading and query operations.

In addition to taking advantage of multiple processors, Red Brick Decision Server automatically partitions the work into multiple processes, thus introducing parallel processing into systems using only a single processor. For example, some queries can be partitioned into multiple processes based on how the data is distributed. While one process is waiting on disk I / O, other processes can proceed, thus increasing the efficiency and speed of query processing.

When needed data has been strategically distributed over multiple drives using Red Brick Decision Server dimensional segmentation, parallel processes can perform disk accesses simultaneously, improving performance significantly.

UNIX

On UNIX, when the needed data resides on a single disk, database server SuperScan technology allows each process to take advantage of data read by the other processes to reduce the disk access delays. ♦

Physical Implementation of Databases

Red Brick Decision Server can support a large number of databases, essentially limited only by system disk space. Typical installations create and use from one to twenty separate databases.

Each database is a self-contained entity containing system tables, control files, and an arbitrary number of user tables. User tables contain the actual data that users access and update in the course of their work. In addition to user tables, a database can contain synonyms (table aliases), views that restrict the presentation of table data to different users, and macros, which define SQL shortcuts for frequently executed operations.

Indexes and Retrieval Strategies

User tables are supported by a variety of indexes. Indexes are invisible to database users but are critical for data integrity and performance. Red Brick Decision Server automatically provides those indexes required for data integrity. The database administrator can create additional indexes to improve performance:

- A STAR index optimizes join processing (STARjoin) between tables related by foreign key references or tables joined over common columns. This index is a join acceleration index for multitable joins. An administrator can create one or more STAR indexes for better performance. When multiple STAR indexes exist, Red Brick Decision Server automatically selects the best STAR index to use for executing each query by applying a cost model.
- A B-TREE index improves performance and ensures data integrity. Red Brick Decision Server automatically creates a B-TREE index on each primary key to ensure uniqueness and foreign key referential integrity. An administrator can improve query performance by creating an additional B-TREE index on any table column or columns that will be constrained in queries. Additional indexes, however, require additional storage space, a time and space trade-off for the database administrator.
- A TARGET index improves performance when queries consist of multiple weakly selective constraints. Performance improvements are two-fold. The queries run faster, and their processing requires less memory.
- TARGETjoin processing uses TARGET or B-TREE indexes on the foreign keys of a referencing (fact) table to perform multitable joins. TARGETjoin processing is complementary to STARjoin processing. A combination of the two technologies offers excellent performance over a wide range of queries.

Segmented Storage

User tables and indexes are each stored in *segments*, which are collections of operating-system files, or physical storage units (PSUs), that provide the physical disk storage required by table and index data.

Segmented storage offers the following advantages for very large databases:

- Allows the administrator to map logical data to physical segments, with dynamic allocation of storage as needed
- Simplifies database loading and updating; particularly useful with time-cyclic data
- Provides the separation of data necessary for parallel query processing

  Parallelism during query processing is limited by the number of segments and PSUs used for a table (both data and indexes).
- Provides locking at the segment level rather than at the table level so that loading and query operations can often proceed simultaneously
- Allows partial functionality of queries when some segments of data or some indexes are not available
- Provides a smaller unit for data recovery in case of media failure

Named and Default Segments

The two types of segments are *named* and *default* segments. Named segments are created explicitly by an administrator with a CREATE SEGMENT statement. Default segments are created automatically by the system for tables and indexes that the administrator does not place in a named segment or segments.

Named segments offer the database administrator extensive control over disk space allocation, file size and placement, and database growth, but they require more effort from the administrator to plan, create, and manage. Using named segments, a database administrator can partition a table horizontally, distributing the data across multiple segments. For example, sales data might be partitioned by time periods, with each time period residing in a separate segment. Control over the individual segments simplifies maintenance tasks and provides better access.

As an example of how segmented storage is used, consider a database that tracks sales data for the previous two years. The data is distributed across segments so that each segment contains one month of data. Because individual segments can be added, loaded, or dropped independently of the rest of the database, each month the administrator adds a segment containing the loaded, indexed data for the current month and drops the segment containing data for the oldest month. An administrator can make these updates without taking the database offline, and the space in the old segment can be used for data for the next month.

Default segments require no specific management. For practical purposes, they are invisible to both the administrator and users. A table or index in a default segment can span multiple segments or files. The PSUs for a default segment are placed in the database directory (or in a default directory specified by the user). For small, static tables or databases, default segments of the default size are often satisfactory. If circumstances change and you require additional control over the size and growth of tables in a default segment, these default segments can be altered and manipulated in the same manner as named segments.

Implementation

Segmented storage is implemented as follows:

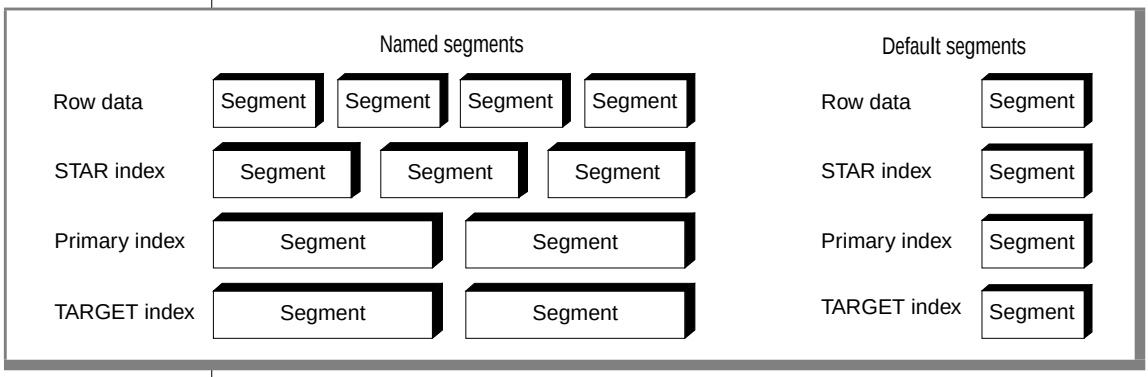
- A segment contains one or more physical storage units (PSUs).
- A PSU can belong at most to one segment.
- A segment can contain either row data for a single table or index data for a single index, but not both.
- Row data and indexes for the table can each span multiple segments.

A segment is located in the directory specified when it was created, in a default location specified in a configuration file or in the directory containing the database.

The following figure illustrates how default and named segments are used.

Named segments are created with a CREATE SEGMENT statement. Segments for row data and automatic indexes are assigned to a table in the CREATE TABLE statement. Segments for optional indexes are assigned in the CREATE INDEX statement.

Figure 2-1
Use of Named and Default Segments



If no segment is assigned to a table or index, the table or index is created in a default segment consisting of one or more PSUs.

PSU Size and Growth

In named segments, each PSU is defined with the following size parameters:

- An initial size, which determines how much space is initially allocated for that file
- A maximum size, which limits how large the file can grow
- An extend size, which defines the size of the increments by which the file grows

By carefully defining these sizes, you can create segments that grow to accommodate additional data and disk space as needed.

In default segments, PSUs have the following characteristics:

- The default initial size is 16 KB.
- The number of PSUs for each segment is determined by dividing the value of the DEFAULT_SEGMENT_SIZE parameter by 2 GB (the initial maximum size of the PSU). The DEFAULT_SEGMENT_SIZE parameter sets the amount of storage for a default table or index.

- The default maximum size of each PSU is 2 GB. The size of the last PSU for a segment is the value of DEFAULT_SEGMENT_SIZE modulo 2 GB. If DEFAULT_SEGMENT_SIZE is not set or is set to less than a single PSU size, the PSU size is 2 GB by default.
- The extend size is determined by the DEFAULT_PSU_EXTENDSIZE parameter. The default extend size is 16 KB.

For example, if DEFAULT_SEGMENT_SIZE is set to 5 GB and DEFAULT_PSU_EXTENDSIZE is set to 512 KB, three PSUs are created, as the following table shows.

PSU	Initial Size	Extend Size	Maximum Size
PSU 1	16 KB	512 KB	2 GB
PSU 2	16 KB	512 KB	2 GB
PSU 3	16 KB	512 KB	1 GB

Distributing Data Among Segments

Row data can be distributed among segments by ranges of values contained in a specified column (the segmenting column) or by hashing. Segmenting by ranges of values offers the advantage of knowing where the data and corresponding indexes reside but can result in an uneven distribution of data. Segmenting by hashing distributes data evenly and prevents *hot spots* but limits the use of offline operations because the location of data is not known. If hashing is used, the entire row is hashed.

A primary index can be segmented based on the key values of the first column of the index, either explicitly specifying ranges of values for each segment or specifying that the index is to be segmented by the same values as the data. (The first column is the column named first in the PRIMARY KEY clause.)

A STAR index can be segmented in two ways. The first method, which is the default, distributes the index entries evenly across the segments, so each segment contains approximately the same number of entries. The second method distributes the index entries across the segments based on the contents of the first column of the STAR index. The first column is the column named first in the CREATE STAR INDEX statement. In a STAR index, the contents of the index entry is not the value of the indexed data but the row ID of the referenced table containing the data.

The following table summarizes the possible ways to distribute the data and indexes among multiple segments, along with the keywords used to specify each type of distribution.

Contents of Segments	How Segmented
Row data	By data value of segmenting column or hashing (keywords: SEGMENT BY VALUES OF, SEGMENT BY HASHING)
STAR indexes	By the rowids of the referenced table containing the segmented column of the STAR index (keywords: none or SEGMENT BY REFERENCES OF)
Primary index	By data values of first (leading) column in the primary index (keywords: SEGMENT BY VALUES OF)
TARGET indexes	By values of the indexed column (keywords: SEGMENT BY VALUES OF)
B-TREE indexes	By values of the first (leading) indexed column (keywords: SEGMENT BY VALUES OF)

Segment boundaries can be modified by attaching or detaching segments at either end of the range of an existing segment, but a new segment cannot be inserted in the middle of the range of an existing segment. The range for any given segment can also be changed (with an ALTER SEGMENT statement), but only if the new range includes all rows already in that segment. The new range cannot require that data be moved.

Example

The following figure illustrates correspondence between the row data segments and the index entry segments for a table with two indexes. This table has a primary index, which is segmented by the data values of the first column in the index, and a STAR index, which is segmented by references to its first column. The index segments of the index are segmented by the same values as the data. The corresponding data and index segments can be taken online or offline together to take advantage of segment operations without interrupting use of the rest of the database. The same correspondence does not exist between the row data segments and segments of the STAR index.

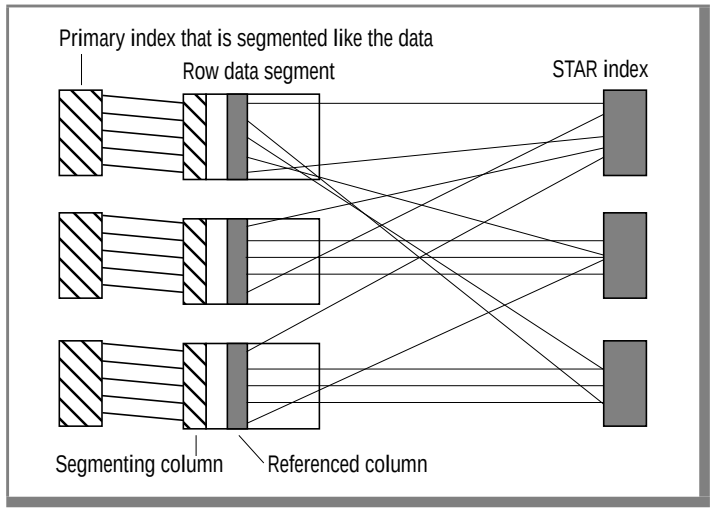


Figure 2-2 Row Data Segment and STAR Index Segments

Online and Offline Segments

Each segment associated with a table or index is either online or offline. When all segments associated with a table and its indexes are *online*, the table is fully available for access. Online is the normal state. When one or more segments associated with a table or its indexes are *offline*, the table is only partially available.

A segment can be taken offline only when it is one of multiple segments associated with a table or index. When a table or index resides in a single segment, that segment cannot be taken offline. Consequently, default segments, which by definition contain an entire table or index, cannot be taken offline.

The administrator can take a segment offline to load or update it with new data, to restore it in case of media failure or other data loss, or to detach it and remove it from the table forever. While that segment is offline, users can still have access to the partially available table.

Partial Availability of Tables and Indexes

Query behavior on a table that is partially available because of offline row data segments is controlled by a SET command with the following options:

- Process all queries and return the results. If the query attempts to access offline segments, issue a warning that the results might be inaccurate, incomplete, or invalid because of offline segments.
- Disallow only those queries that attempt to access offline segments of a table. Other queries on the table return results.
- Disallow all queries on a table that has offline segments.

Query behavior on a table that is partially available because of offline index segments is also controlled by a SET command that determines whether to consider all indexes or only fully available indexes in selecting a query-processing strategy. For more information about controlling query behavior with partially available tables, refer to "Query Behavior on Partially Available Tables" on page 10-25 and "Use of Partially Available Indexes" on page 10-28.

No operation that modifies the contents of a table or index—except an offline load operation—is allowed on a partially available table. These operations include load, insert, update, and delete operations, and also creating and dropping indexes. A delete operation on a table that is referenced by a foreign key of a partially available table is not allowed.

Precomputed Views for Increased Query Performance

Aggregate tables can dramatically increase query performance for large databases. You can define precomputed views so that queries are automatically rewritten to use the best aggregate table available. A precomputed view definition includes:

- A physical table containing precomputed aggregate values
- A logical view that allows the query rewrite engine to be aware of the precomputed table and then rewrite queries that are submitted against a detail-level table (for example, **Daily_Sales**) to a precomputed table (for example, **Monthly_Sales**)

Additionally, you can define logical rollup hierarchies that allow queries that do not exactly match the data in the precomputed (aggregate) table to be rewritten against the precomputed table.

When planning your physical implementation of the database, you must plan for the precomputed (aggregate) tables you intend to create. These aggregate tables are just like any other tables. Therefore, they can and should be segmented and indexed. They often contain primary key / foreign key definitions, so STAR indexes can be defined to enable STARjoin processing, and TARGET and B-TREE indexes can be defined on the foreign keys to enable TARGETjoin processing.

Vista provides an aggregate maintenance service that automatically updates aggregate tables when their detail tables are updated. Aggregate maintenance ensures consistency between detail and aggregate tables without user-defined update scripts.

Vista also includes an Advisor, which stores query history in log files. The Advisor uses these log files to keep track of how often the current precomputed views in your database are being accessed, as well as to suggest the ideal set of precomputed views to create based on the performance benefit they would provide.

For detailed information on using Vista, refer to the *Informix Vista User's Guide.*

Query Sampling for Application Input

Red Brick Decision Server provides a sampling feature to run queries that operate on a *random* sample of rows instead of the full data set. The result set for a query that uses sampling is representative of the complete result set. Sampling query results can provide the input to another application—such as a *training set* for a test database or data passed to a data mining program. In this context, the sampling feature is more of an application enhancement than a performance optimization. Sampling is not likely to optimize performance for queries (or subqueries) unless they require a table scan to return the full result set. For queries that do not require table scans, using the sampling function might degrade performance.

In the case of candidate view analyses performed with the Vista Advisor, constraining on a sample of the fact table rows does optimize performance. Informix recommends the use of the default ADVISOR_SAMPLE_SIZE parameter to optimize candidate analyses, as described in the *Informix Vista User's Guide*.

For more information on query sampling, see the *SQL Reference Guide.*

Database Directories and Files

A database is initially created in a single directory. This directory, termed the *database directory*, contains control files and system tables. With default segments, the database directory contains all of the files with user tables and indexes. With named segments, however, these files can reside in additional directories that are not subdirectories of the database directory or not even in the same file system.

The database directory and other directories containing named segments can be located anywhere in the operating-system file space (that is, on any file system and located anywhere in the directory structure of that file system).

All directories and files relating to a database and its segments must be owned by the **redbrick** user. For maximum security, read or write access should not be extended to any other user or group. Access to the contents of the database and to tablespace directories and files is controlled by Red Brick Decision Server.

Logical Database Names

Users specify a logical database name for the database they want to access. Red Brick Decision Server automatically translates this logical name to the appropriate pathname. Mappings between logical database names and database directory paths are defined by the administrator and stored in the **rbw.config** file.

The following examples illustrate the contents and organization of typical database directories.

Example

Assume that a database is created with the following statements using default segments. The database directory is **newdb**, located in the path **/warehouse/mktg/newdb** on UNIX and **c:\warehouse\mktg\newdb** on Windows.

```
CREATE TABLE period (
   ...
   PRIMARY KEY (perkey));
CREATE TABLE product (
   ...
   PRIMARY KEY (prodkey));
CREATE TABLE market (
   ...
   PRIMARY key (mktkey));
CREATE TABLE fact1 (
   ...
   PRIMARY KEY (perkey, prodkey, mktkey)
   FOREIGN KEY (perkey) REFERENCES period (perkey)
   FOREIGN KEY (prodkey) REFERENCES product (prodkey)
   FOREIGN KEY (mktkey) REFERENCES market (mktkey));
```

The following figure illustrates the contents of the directory containing this database, which uses default segments.

With default segments, each table and index is stored in a separate segment, each segment initially consisting of a single PSU. If the table outgrows a single PSU, you can add more PSUs as needed to hold the data.

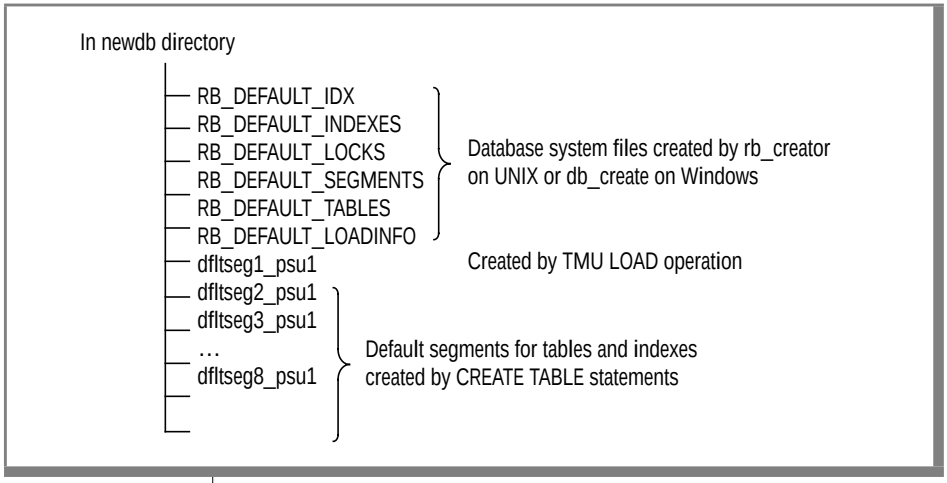


Figure 2-3
Directory of Logical Database Names

Example

Assume that the same database is created in the **newdb** directory, but segments are created to hold the table **Fact1** and its indexes. Data in this table is segmented by time period, putting 1999 data in one segment and 2000 data in another segment. The primary key index is created and placed in a third segment.

UNIX

```
CREATE SEGMENT fact_99
   STORAGE '/disk1/fact_99/99_dat1'
      MAXSIZE 1000 initsize 100 extendsize 100,
   STORAGE '/disk1/fact_99/99_dat2'
      MAXSIZE 1000 initsize 100 extendsize 100;
CREATE SEGMENT fact_00
   STORAGE '/disk2/fact_00/00_dat1'
      MAXSIZE 1000 initsize 100 extendsize 100,
   STORAGE '/disk2/fact_00/00_dat2'
      MAXSIZE 1000 initsize 100 extendsize 100;
CREATE SEGMENT fact_pi
   STORAGE '/disk2/fact_pi/fact_pi1'
      MAXSIZE 100 initsize 20 extendsize 10,
   STORAGE '/disk2/fact_pi/fact_pi2'
      MAXSIZE 100;
CREATE TABLE market (
   ...
   PRIMARY key (mktkey));
CREATE TABLE product (
   ...
   PRIMARY KEY (prodkey));
CREATE TABLE period (
   ...
   PRIMARY KEY (perkey));
CREATE TABLE fact1 (
   ...
   PRIMARY KEY (perkey, prodkey, mktkey)
   FOREIGN KEY (perkey) REFERENCES period(perkey)
   FOREIGN KEY (prodkey) REFERENCES product (prodkey)
   FOREIGN KEY (mktkey) REFERENCES market (mktkey)
) DATA IN (fact_99, fact_00)
      SEGMENT BY VALUES OF (perkey)
      RANGES (MIN:'1998-12-31', '2000-01-01':MAX)
   PRIMARY INDEX IN fact_pi;
```

The directories **/disk1/fact_99, /disk1/fact_00,** and **/disk2/fact_pi** must exist before the CREATE SEGMENT statements can be issued. However, the segment name and directory name need not be identical. ♦

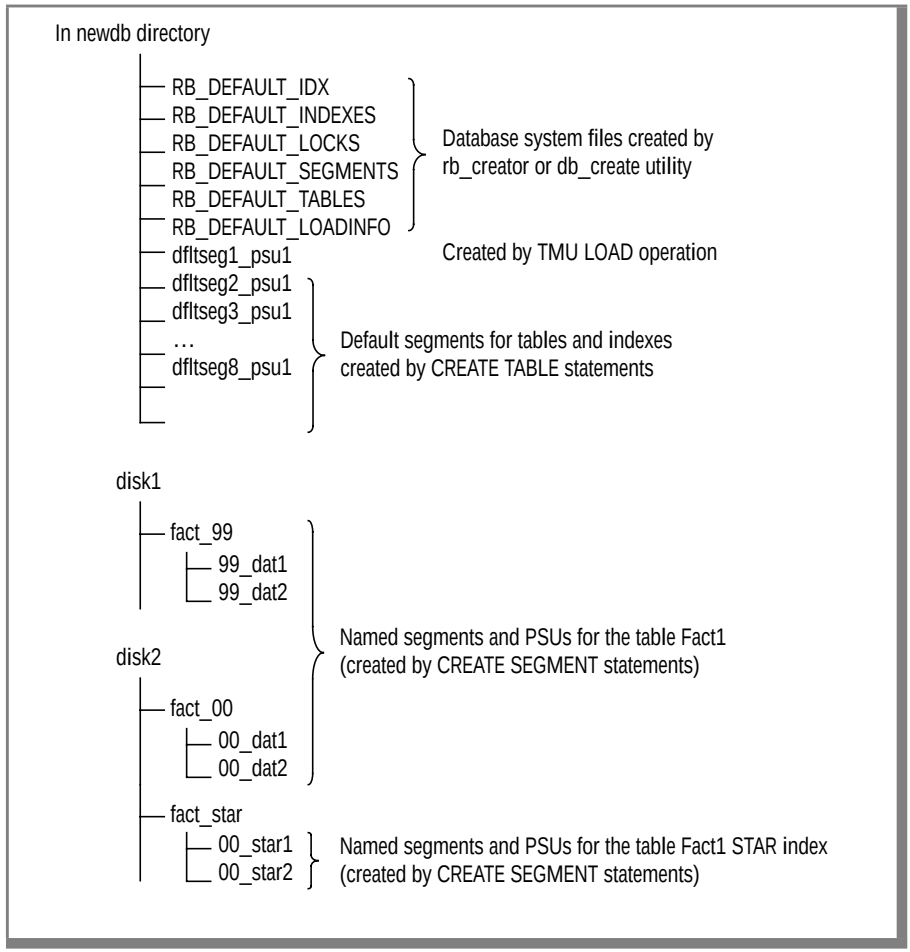
WIN NT/2000

```
CREATE SEGMENT fact_99
   STORAGE 'c:\disk1\fact_99\99_dat1'
      MAXSIZE 1000 initsize 100 extendsize 100,
   STORAGE 'c:\disk1\fact_99\99_dat2'
      MAXSIZE 1000 initsize 100 extendsize 100;
CREATE SEGMENT fact_00
   STORAGE 'c:\disk2\fact_00\00_dat1'
      MAXSIZE 1000 initsize 100 extendsize 100,
   STORAGE 'c:\disk2\fact_00\00_dat2'
      MAXSIZE 1000 initsize 100 extendsize 100;
CREATE SEGMENT fact_pi
   STORAGE 'c:\disk2\fact_pi\fact_pi1'
      MAXSIZE 100 initsize 20 extendsize 10,
   STORAGE 'c:\disk2\fact_pi\fact_pi2'
      MAXSIZE 100;
CREATE TABLE market (
   ...
   PRIMARY key (mktkey));
CREATE TABLE product (
   ...
   PRIMARY KEY (prodkey));
CREATE TABLE period (
   ...
   PRIMARY KEY (perkey));
CREATE TABLE fact1 (
   ...
   PRIMARY KEY (perkey, prodkey, mktkey)
   FOREIGN KEY (perkey) REFERENCES period(perkey)
   FOREIGN KEY (prodkey) REFERENCES product (prodkey)
   FOREIGN KEY (mktkey) REFERENCES market (markey)
) DATA IN (fact_99, fact_00)
      SEGMENT BY VALUES OF (perkey)
      RANGES (MIN:'1999-12-31', '2000-01-01':MAX)
   STAR INDEX IN fact_pi;
```

The directories **c:\disk1\fact_99, c:\disk1\fact_00,** and **c:\disk2\fact_pi** must exist before the CREATE SEGMENT statements can be issued. However, the segment name and directory name need not be identical. ♦

The following figure illustrates how the database files are stored using named segments.

Figure 2-4
Example of Segmentation Schema of Database System Files



Segment Names

The **RBW_SEGMENTS** system table contains the segment names of each table, as the following example shows:

```
select name, tname, iname from rbw_segments;

NAME                  TNAME      INAME
RBW_SYSTEM            NULL       NULL
DEFAULT_SEGMENT_1     PERIOD     NULL
DEFAULT_SEGMENT_2     PERIOD     PERIOD_PK_IDX
DEFAULT_SEGMENT_3     PRODUCT    NULL
DEFAULT_SEGMENT_4     PRODUCT    PRODUCT_PK_IDX
DEFAULT_SEGMENT_5     MARKET     NULL
DEFAULT_SEGMENT_6     MARKET     MARKET_PK_IDX
FACT_99               FACT1      NULL
FACT_00               FACT1      NULL
FACT_STAR             FACT1      FACT1_STAR_IDX
```

Configuration and Initialization

Red Brick Decision Server uses configuration and initialization files, SET commands, and environment variables to customize each database server installation. Combinations of file settings and interactive SET commands can be used to provide varying degrees of global, database, and user customization.

Configuration File

Database server configuration information is contained in the **rbw.config** file, which resides in the database server directory. This file is generated during the installation process and contains configuration and performance-tuning parameters used by the warehouse daemons, database server, and TMU processes on UNIX and by the Red Brick Decision Server service and TMU on Windows. It can be edited with a text editor.

Initialization Files

Red Brick Decision Server uses three types of initialization files: **.rbwrc** files for database server (**rbwsvr**) initialization, **.rbretrc** files for RISQL Entry Tool and RISQL Reporter initialization, and Red Brick ODBC Driver initialization files, which contain definitions for Red Brick ODBC Driver applications. (Red Brick JDBC Driver does not have initialization files because the JDBC URL provides the machine name and port.)

.rbwrc Files

The **.rbwrc** files are used by the database administrator to customize each user session according to specific user needs. These files can contain any SQL statements that are not queries, such as CREATE MACRO, INSERT and SET. File permissions must be set so that the database server process, which runs as the **redbrick** user, can read the **.rbwrc** files and write to the **.rbwerr** file.

Up to four **.rbwrc** files determine a user profile. These files are processed in the following order.

UNIX File	Windows File	Description
$RB_CONFIG/.rbwrc	%RB_CONFIG%\.rbwc	A global warehouse file in the directory that contains **rbw.config**.
$RB_PATH/.rbwrc	%RB_PATH%\.rbwrc	A database-specific file in each database directory.
$HOME/.rbwrc	None	A user-specific file in each user's home directory for users who access databases with RISQL Entry Tool or RISQL Reporter running on the same computer as Red Brick Decision Server.
$RB_PATH/.rbwrc.DBUSERNAME	%RB_PATH%\.rbwrc.DBUSERNAME	A user-specific initialization file. If used, this file must reside in the database directory with the following extension: `.rbwrc.`*`DBUSERNAME`* *DBUSERNAME* is the database user name, not the operating-system user account name. The extension must be uppercase.

The processing order allows database settings to override database server settings, and user settings to override both database and database server settings.

The database server and database initialization files are managed by the database administrator. These files generally contain global or database-specific temporary macros, access-control statements, values that limit the size of database working files, and directives for placement of null values in ORDER BY clauses.

User-specific files can be edited and changed by individual users. These files are generally used for temporary macros and for SET commands to customize a user session.

The commands in these files, along with any error, informational, or warning messages they generate, are echoed to a log file named **.rbwerr** in the user's home directory. Each time the database server process (**rbwsvr**) is started, these log files are deleted to prevent their unlimited growth.

.rbretrc Files

The **.rbretrc** files customize the execution profile of a user accessing the database server with the RISQL Entry Tool and RISQL Reporter. These files contain SET commands. For example, a command specifies the editor of choice or specifies display widths for system table columns to format the screen displays. Two **.rbretrc** files, processed in the following order, can affect a user.

UNIX File	Windows File	Description
$RB_CONFIG/.rbretrc	%RB_CONFIG%\.rbretrc	A file in the directory that contains **rbw.config**. This file initializes the session of any user who accesses the database server.
$HOME/.rbretrc	%HOMEPATH%\.rbretrc	A user-specific file in each user's home directory. It affects a single user profile.

The processing order allows user settings to override database server settings. Command-line settings override settings in the **.rbretrc** files but exist only for the current session.

ODBC Initialization Files

ODBC Data Source Name (DSN) definitions are used by client applications (for example, RISQL Entry Tool or RISQL Reporter) to connect to Red Brick Decision Server through the Red Brick ODBC Driver. A DSN is a logical name that defines a server, a logical database name, a database user name, and, optionally, a database password.

UNIX

On UNIX, if the **ODBCINI** environment variable is set, the DSN definitions are taken from the file specified in the **ODBCINI** environment variable; otherwise, the DSN definitions are taken from a user-specific **.odbc.ini** file in the user's home directory. Installation of any Client Connectivity component creates a template file named **odbc.ini** in the **redbrick** directory. From this template file an administrator can create a shared DSN definitions file accessed by all users by setting the **ODBCINI** environment variable to that filename. Alternatively, each user can create a user-specific **.odbc.ini** file that resides in their home directory. ♦

WIN NT/2000

On Windows, use the ODBC Data Source Administrator in the Control Panel to add or change DSNs. The **rbodbc32.ini** file in the Windows operating system directory contains initialization information for the Red Brick ODBC Driver. Use this file to set the default locale of all clients on the system. ♦

For more information, see the *Client Installation and Connectivity Guide*.

SET Commands

SET commands are used to configure and customize the database server, the TMU, and the RISQL Entry Tool and RISQL Reporter. These commands can be specified in several ways, depending on the command:

- In either the **.rbwrc** or **.rbretrc** initialization file
- In a TMU control file
- From the RISQL Entry Tool or RISQL Reporter
- From a client tool sending SQL statements

Environment Variables

The database server uses the following environment variables.

Environment Variable	Description
RB_CONFIG	Pathname to the directory containing the **rbw.config** file, which is usually the directory containing the **bin** directory of database server executable files.
RB_DSN	Defines data source name (DSN). For more information, refer to the *Client Installation and Connectivity Guide*.
RB_EXE (Windows)	Name of the Red Brick Service executable file, set to **service.exe** by default.
RB_HOME (Windows)	Pathname of the home directory where Red Brick Decision Server is installed.
RB_HOST	Logical name used to identify the warehouse daemon process on UNIX or the Red Brick Decision Server service on Windows.
RB_PATH	Logical database name, as defined in **rbw.config** file by a pathname to a database directory. Used to determine which database to access.

Administrator Tool

The Red Brick Decision Server Administrator tool provides a graphical interface for database administration. It is a stand-alone application that runs under the Windows operating systems. It can access databases running on either UNIX or Windows. The Administrator tool provides direct access to multiple databases for database administration tasks, including:

- Create, alter, and drop users, roles, macros, tables, indexes, segments, synonyms, and views
- Perform detailed segment-related tasks such as examining and verifying PSUs, attaching and detaching segments, and defining PSU attributes
- Grant and revoke privileges and authorizations for users and roles

- Perform general database tasks such as quiescing, resuming, resetting the administration daemon, and resetting statistics on the database
- Control user activity and change the priority of a current user session
- Manage database logging operations
- Manage database accounting activity
- Determine the actual and maximum size of tables and indexes in a database

Additionally, the Administrator tool allows you to graphically view:

- The relationships between referenced and referencing tables in the database
- The structure of your file systems
- Information about the data, index, and unattached segments in your database
- The structure of your database, showing users, roles, macros, and data objects (tables, system tables, views, and synonyms)
- Property information about each data object, gathered from the relevant system tables
- Query plans generated by the EXPLAIN command
- Utilization and candidate view analyses generated by Vista Advisor

The Administrator tool also provides an interactive SQL window that allows you to enter SQL statements manually and a Show DDL window in which you can view the SQL statements that were used to create a selected database object.

Server Locale

This section defines the term *locale*, describes how the server locale is specified during installation, explains how to override the locale for a client tool, discusses some compatibility issues that might arise in a client/server environment, and explains the effect of the locale specifications on the message system. It includes the following sections:

- Components of a Locale
- Defining the Server Locale
- Overriding the Server Locale
- Ensuring Client/Server Compatibility

Components of a Locale

The unique combination of a language and a location is known as a *locale*. The server locale, defined during installation, is the administrator's mechanism for controlling the runtime behavior of databases. A locale specification consists of four components: *language*, *territory*, *code set*, and *collation sequence*. For example:

```
Japanese_Japan.MS932@Binary
```

- `Japanese` = language
- `Japan` = territory
- `MS932` = code set
- `Binary` = collation sequence

The following sections explain each locale component in detail. For a list of supported locales, see the **locales.pdf** file in the **relnotes** directory on the Red Brick Decision Server product CD-ROM. For detailed information about specifying a server locale, refer to "Defining the Server Locale" on page 2-31.

Language

The language component (in conjunction with the territory) controls which translation is used. In general, text strings are accepted and displayed in the user's chosen language. These strings include information and warning messages, object names, month and day names, and character data returned in query results. However, the fixed elements of a programming language, such as the keywords used in SQL statements, are not translated.

Territory

The territory component controls country-dependent information such as currency symbols, numeric and monetary formatting rules, and date and time formats. For example, although English is used in both the United States and the United Kingdom, and Spanish in both Spain and Mexico, these languages differ according to the location. (In some cases, a single territory applies to more than one country in a region.)

The formatting rules for various noncharacter data types vary around the world. For example, in much of Europe, the decimal point in a floating-point number is represented by a comma. However, in some European countries and the United States, the decimal point is represented by a period. Similarly, the thousands separator in numbers varies, and numbers are not separated into groups of 1,000 in some parts of the world.

The rules for formatting dates are also subject to local convention. The order of the components of a date (year, month, day), the character separating the components, the names of the components, how they are abbreviated, and even the calendar can all vary by location. Time can be represented based on either a 12-hour or a 24-hour clock, and the Latin labels A.M. and P.M. change for different languages. The formatting rules for currencies also vary widely, primarily in the placement of the currency symbol.

Code Set

The code set component specifies the character encoding scheme or code page used to format and display information. A code set specifies how a set of characters used by one or more languages is mapped to a somewhat arbitrary set of numbers. These numbers are referenced when keyboard input is converted to information displayed on the screen. For a character to be recognizable, every system that processes it must use the same encoding.

In the United States, a 7-bit encoding known as ASCII is commonly used. However, this encoding is inadequate for international use because all 128 encodings are assigned, and they do not include all the characters necessary to represent languages other than English. Therefore, Red Brick Decision Server also supports the following character encodings:

- 8-bit encodings, which use the eighth bit of the byte to extend ASCII and can represent 256 characters. Eight-bit encodings are adequate for most European languages and have the advantage of preserving the property that a byte and a character are the same size.
- Multibyte encodings, which allow the width of a character to vary from 1 to 4 bytes. Multibyte encodings contain 7-bit ASCII as a subset. The Asian languages (Japanese, Chinese, Korean), which require significantly more than 256 characters, typically use a multibyte encoding.

For a list of supported code sets and supported conversions between code sets, refer to the **locales.pdf** file in the **relnotes** directory on your Red Brick Decision Server product CD-ROM.

Collation Sequence

The sort component of the locale, or collation sequence, defines the rules used to compare character strings and arrange them in the correct order. The two main types of character comparisons are *binary* and *linguistic*.

Binary Character Comparisons

In a binary comparison, the character encoding assigned to each character is interpreted as a number, and the numbers are compared and sorted. If the number for one character is less than the number for another character, the first character precedes the second in the collation sequence.

Although binary comparisons are relatively fast, they do not always yield useful results. Binary comparisons are adequate for 7-bit ASCII and English because the binary encodings for the letters *a* to *z* appear in the correct order. However, the following limitations apply:

- All uppercase letters sort before any lowercase letters. For example, the lowercase letter *a* sorts after the uppercase letter *Z*.
- When an 8-bit ASCII encoding is used, characters with encodings in the range 128 to 255 do not sort between characters with encodings less than 127. For example, all the vowels with diacritical marks used in European languages appear in the upper half of the 8-bit ASCII encoding, but most are supposed to sort together with the unmarked vowel.
- For some languages, a binary collation sequence is never accurate because the sort position of a character might depend on the character that follows.

Linguistic Character Comparisons

The solution to the limitations of binary comparisons is a *linguistic* or *lexical* character comparison, which takes into account the customary sorting rules associated with a language. Any given language might use more than one set of sorting rules. For example, in some countries, the names in a telephone directory are sorted differently from the words in a dictionary. Even though the rules are different, both methods are considered linguistic because the sort order has nothing to do with the binary character encoding.

A linguistic comparison is often performed by retrieving an entry from a table for a particular character to be compared. This entry indicates where the character resides in the sort sequence and can be compared to other entries from the same table to determine its relationship to other characters. A linguistic sort might also need to be context sensitive (where a character sorts might depend on what characters precede or follow it). Because of the table lookups and context sensitivity, linguistic comparisons are relatively slow compared to binary comparisons.

In summary, binary comparisons are less flexible but yield better performance, whereas linguistic comparisons do not perform as well but return more meaningful results in international markets. Red Brick Decision Server supports both types of comparisons, as defined in the locale specification during installation.

Defining the Server Locale

During the installation of the Red Brick Decision Server software, a locale specification is requested for the database server. The locale supplied during installation is stored as the NLS_LOCALE LOCALE parameter in the **rbw.config** file. If no locale is supplied, the default value of this parameter is used:

```
English_UnitedStates.US-ASCII@Binary
```

This locale specification applies to the whole Red Brick Decision Server installation, regardless of the number of databases that are created for that installation. (An *installation* is defined by the contents of the *rbw.config* file found in the directory referenced by the **RB_CONFIG** environment variable.) This restriction means, for example, that all character columns in each database are stored using the same character encoding and that all indexes are sorted according to the same collation sequence.

A locale is both an attribute of stored data—a database, a TMU input file, or a backup tape—and a configuration parameter for the database server products that regulates their runtime behavior. In most cases, the locale defined during installation and the operating locale of those products must be the same. The exceptions to this rule are described in this chapter.

All nonclient products, including Red Brick Decision Server, the TMU, and the miscellaneous utility programs, use the locale defined in the **rbw.config** file to determine their runtime behavior. For example, if the language is set to Japanese, all database server and TMU error messages are displayed in Japanese. Similarly, if the locale specifies a multibyte code set such as MS932, database object names and character strings can contain multibyte characters.

For full instructions about defining the server locale during installation, refer to the *Installation and Configuration Guide*.

System Table References to Locales

The server locale and the current client locale are stored in the DB_LOCALE and NLS_LOCALE rows, respectively, of the **RBW_OPTIONS** system table.

Nontranslated Text

Regardless of the locale specified for a database server installation, the following text always appears in English:

- Installation scripts
- Contents of the **rbw.config** file

  All text in this file must be in ASCII characters.
- Contents of system tables (except for object names)
- Log messages
- Output of the EXPLAIN statement (except for object names)
- Output of all Red Brick Decision Server utility programs (**rb_creator** or **dbcreate**, **rb_deleter**, **dbsize**, and so on).

  However, these utilities can handle multibyte characters in data and object names as well as perform comparisons according to the collation sequence of the server locale. Also, data formatting is not localized.

Overriding the Server Locale

In the client/server environment, the operating locale of the client can differ from the server locale defined in the **rbw.config** file. Therefore, the locale defined for a client tool can differ from the locale of the database server. However, only differences in language and code set are of practical value, and no warning is given when the client and server locales do not match.

Changing the language controls the language in which messages are displayed, and changing the code set allows the client and database server to use different code sets, as long as a successful conversion can be made.For a list of supported code sets for each language, refer to the **locales.pdf** file in the **relnotes** directory on your Red Brick Decision Server product CD-ROM.

Changing the territory has little effect because territory primarily controls display formatting, which the database server does not do. Similarly, changing the sort component has no effect because all sorting is done by the database server according to the collation sequence defined for the database.

For information on overriding the server locale with the TMU, refer to the *Table Management Utility Reference Guide*.

Specifying a Locale for a Client Tool

A client tool can run in its own locale rather than that of the database. This client locale is used to format output and to identify the correct message file. Messages displayed through the client might be generated by the client or the database server. Regardless of where they are generated, all messages are displayed in the language of the client locale.

The client and database server can use different code sets, as long as those code sets support the same language and can successfully be converted. For example, MS932 Japanese characters can be converted to JapanEUC Japanese characters.

The RISQL Entry Tool and RISQL Reporter are Informix client applications. An operating locale can be set for each of these clients by setting the **RB_NLS_LOCALE** environment variable. If no client locale is specified, the behavior depends on the operating system.

UNIX

On UNIX, clients use the server locale, as specified by the current NLS_LOCALE LOCALE entry in the rbw.config file. ♦

WIN NT/2000

On Windows, the client locale is specified in the **rdodbc32.ini** file in the Windows operating system directory. ♦



Tip: *Third-party client tools must provide their own means of specifying a locale, but they can still use the* ***RB_NLS_LOCALE*** *environment variable to control code set conversions and the locale of the messages generated by the database server.*

Setting the RB_NLS_LOCALE Environment Variable

The **RB_NLS_LOCALE** environment variable regulates the locale for the current user. It is the only means by which RISQL Entry Tool or RISQL Reporter users can specify which language they want to use when they start the application.

UNIX

For example, from the UNIX C shell prompt:

```
%setenv RB_NLS_LOCALE German_Austria.Latin1@Default
```

♦

WIN NT/2000

For example, from the MS-DOS shell prompt:

```
c:\> set RB_NLS_LOCALE=German_Austria.Latin1@Default
```

♦

In these examples, the variables are as follows:

```
German = language
Austria = territory
Latin1 = code set
Default = collation sequence
```

You need not specify all four parts of a locale. However, the language should be one of the specified components. Otherwise, the unspecified components might default to incompatible values.

The following rules apply to default values for unspecified locale components:

- If only the language is specified, the omitted components are set to the default values for that language. For example, if the locale is set to `Japanese`, the complete locale specification is as follows:

  ```
  Japanese_Japan.JapanEUC@Binary
  ```

  For a list of default components for each language, refer to the **locales.pdf** file in the **relnotes** directory on your Red Brick Decision Server product CD-ROM.
- If only the territory is specified, the language defaults to `English`, the code set to `US-ASCII`, and the sort to `Binary`. For example, if the locale is set to `_Japan`, the complete, but *impractical*, locale specification is as follows:

  ```
  English_Japan.US-ASCII@Binary
  ```
- Similarly, if only the code set is specified, the language defaults to English, the territory defaults to `UnitedStates`, and the sort component defaults to `Binary`.
- Finally, if only the sort component is specified, the language defaults to `English`, the territory defaults to `UnitedStates`, and the code set defaults to `US-ASCII`.



Tip: *You need not specify all the separator characters (the underscore, the period, and the @ character) in a partial locale specification. Only the character that immediately precedes the component is required, such as the underscore character (_) in the previous territory example.*

Ensuring Client/Server Compatibility

Internationalization raises some potential compatibility issues in the client / server environment. Consider these scenarios:

- A Japanese user working on a PC might set the client locale to:

  ```
  Japanese_Japan.MS932
  ```

  However, the database server might be using the JapanEUC code set instead of MS932. If so, the database server must perform a code set conversion on the processed data before sending it to the client PC. Because this code set conversion is supported, no data loss or incompatibility should arise.

- A French-speaking user might connect to a database with a German locale:

  ```
  French_France.Latin1 (client locale)
  German_Germany.Latin1 (warehouse locale)
  ```

  In this case, no code set conversion is required because German and French both rely on the Latin1 code set. However, the user's client tool will display messages in French as well and use French formatting rules for datetime data. Nonetheless, the data itself is assumed to be German and will follow German sorting rules, regardless of the sort component specified for the client.

- An English-speaking user might attempt to use a Japanese database:

  ```
  English_UnitedStates.US-ASCII (client locale)
  Japanese_Japan.JapanEUC (warehouse locale)
  ```

  Multibyte data queried by the user will not be correctly displayed although the database server will accept ASCII data in load and insert operations. The client and server locales are incompatible.

The following sections explain how to avoid and correct certain client / server incompatibilities.

Code Set Conversions

Before configuring a client / server environment to use different code sets for the database and client locales, make sure conversion between those code sets is supported.



Warning: *Red Brick Decision Server provides no recovery mechanism when data loss or data corruption occurs because of incompatible code sets.*

Message System

The message system should display error, warning, and information messages to users in the appropriate language. To accomplish this, the system maintains a separate message file for each supported language.

In the client/server environment, the locale of the client determines which message file is used. In all other cases, the server locale determines the message file. The language and territory components of the locale specification control which message file is selected. In turn, messages are displayed in the appropriate language, and the text follows the appropriate regional conventions.

If the message file for the requested locale cannot be found, the default U.S. English message file is used, without warning. If that message file cannot be found, an error message is issued. This message is always in English.

Naming Convention for Message Files

To allow multiple message files in different languages to exist for a single installation, the following filename formats are used:

- Database server error messages: **RBL*LTTT*.MB**
- Database server log messages: **RL*LLTTT*.MB**
- Client messages: **RC*LLTTT*.MB**

LL identifies the language.

TTT identifies the territory.

All letters in message filenames are uppercase. For example, the database server message file for U.S. English is named **RBENUSA.MB**.

The location of the message files is indicated by the NLS_LOCALE MESSAGE_DIR parameter in the **rbw.config** file.

Internal Error Messages

Internal error messages are not stored in the message files. These messages are always displayed in U.S. English.

Output of the Administration and Log Daemons

The administration (**rbwadmd**) and log (**rbwlogd**) daemons or threads generate information that tracks database activity. Although this information is not translated, the output of these daemons might contain data that was generated by the database server and is therefore in the language of the server locale.

For more information about the administration and log daemons, refer to Chapter 8, "Managing Database Activity in an Enterprise."

File Ownership and Permissions

In the database server installation procedure, you create an operating-system user account. The default name for this account is **redbrick** although you can choose any name. Throughout the Red Brick Decision Server documentation, this user account is referred to as **redbrick**, and this user ID is used for all database administration activities at the operating-system level.

All software is installed and owned by **redbrick**, with permissions set for the necessary access. Administrative activities such as creating, deleting, and loading databases are performed with utilities that can be run only by the **redbrick** user. All database files and directories, including segments, system and user tables, indexes, and configuration files, are owned by **redbrick**. Read access should not be extended to other users or groups. Users gain access to these files only through the database server interfaces and never read or modify database files directly.

Database Authorizations and Privileges

In addition to the **redbrick** account used for operating-system activities, the database administrator uses a DBA account in each database to grant users database access. This account exists on each new database with user name **system** and password **manager** and has membership in the DBA system role. From this account, the database administrator uses GRANT statements to set up user access to each database.

The database administrator grants each user database access with a GRANT statement that makes the user a member of a predefined system role. Red Brick Decision Server contains three predefined system roles.

System Role	Description
CONNECT	Authorization to access a database. A member who belongs only to the CONNECT system role cannot create or drop database objects.
RESOURCE	Authorization to create and delete tables, indexes, and views and to grant and revoke privileges on those tables and views. The RESOURCE system role includes the authorizations of the CONNECT system role.
DBA	Authorization to grant and revoke DBA, RESOURCE, or CONNECT system roles and perform administration tasks. The DBA system role includes the authorizations of the RESOURCE and CONNECT system roles.

Object privileges are granted on a table-by-table basis for a specific task (SELECT, INSERT, UPDATE, DELETE) or for all tasks (ALL PRIVILEGES).

Object privileges can be granted to PUBLIC (all members of the CONNECT system role) or to individual users.

The database administrator can use role-based security features to control database access with a finer degree of detail. These features allow you to break the tasks of the RESOURCE and DBA system roles into separate task authorizations and to create custom roles.

For more information about system roles, privileges, and role-based security, refer to Chapter 7, "Providing Database Access and Security," and to the *SQL Reference Guide*.

Versioned Databases

Versioning is a mechanism whereby *readers*, typically users querying the database, can access a verified version of the database while *writers* (for example, INSERT, UPDATE, DELETE, REORG, and LOAD operations) create a new version of the database with little or no impact on reader operations. You can choose to provide the latest version of the database to user queries or to provide the same version to all queries to ensure consistent results in data analysis. An operation that is not versioning is referred to as blocking.

A *transaction* in a Red Brick Decision Server database is defined as a single executable statement. There is no syntax for BEGIN TRANSACTION and COMMIT, but they are implicit at the beginning and end, respectively, of every SQL statement and every TMU operation executed. The duration of the transaction is the time the statement takes to fully execute.

A *versioned transaction* is a transaction on a versioned database that changes the database (for example, INSERT, UPDATE, DELETE, and LOAD operations). Versioned transactions create new versions of the blocks they modify. These new versions reside in the version log until they are written back to the database files. Red Brick Decision Server provides two types of versioning: one for concurrency and the other for recoverability.

Versioning for concurrency is effective for a decision support system (DSS), which is designed to perform complex analyses on large amounts of data. In contrast, OLTP systems are designed to favor the processing of multiple small transactions and use more complex concurrency models that favor data manipulation over analysis. Decision support systems are designed to perform complex analyses on large amounts of data and process queries that often access large numbers of rows in many tables. These complex queries are where the real value of a DSS database is realized, so the concurrency model in a DSS environment favors query performance over update performance.

Versioning for recoverability uses the version log as a traditional write-ahead log. The high recoverability option requires fewer resources for the version log and has a smaller time window during which data can be lost than when using versioning for concurrency. Versioning for recoverability is suitable for large nightly loads that have exclusive access to the table. It does not allow for concurrent queries.

For more information, refer to Chapter 6, "Working with a Versioned Database."

Referential Integrity

Referential integrity is the relational property that each foreign key value in a table exists as a primary key in the referenced table. Red Brick Decision Server requires that referential integrity be maintained in order to produce valid query results and also to build its STAR indexes. Referential integrity relationships are defined with SQL FOREIGN KEY and PRIMARY KEY clauses in the CREATE TABLE statement and are automatically maintained both during load, update, and insert operations to a referencing table and during delete operations from a referenced table.

Load and Insert Operations

During a load or insert operation, if a row is to be added to a table and that row contains a value in a foreign key column that is not present in the table referenced by the foreign key, adding the row would violate referential integrity, so the row is discarded. On load operations, these discarded records can be saved to a file and reloaded later, after a new row containing the missing foreign key value is inserted into the referenced table or after the file is edited to correct data conversion or content errors.

An alternative to discarding rows that violate referential integrity is generating a row with the new value and adding it to the referenced table, thereby preserving referential integrity. The new row is filled in with default values defined when the table was created. This alternative behavior is implemented by a TMU option named Automatic Row Generation (AUTOROWGEN).

Delete Operations and Cascaded Deletes

During a delete operation, if a row to be deleted contains a value that is referenced by a foreign key in another table, a referential integrity violation is avoided by either:

- Deleting the original row and also deleting the referencing row from the other table. This action is called a cascaded delete and can cascade through a series of referencing tables.
- Deleting neither row—that is, taking no action. This lack of action is called a restricted delete.

The course of action to be taken—a cascaded or restricted delete—is specified at the time the table is created by the values CASCADE or NO ACTION in the ON DELETE clause of the CREATE TABLE statement.

During the delete operation, the system must lock not only the table from which the row is being deleted but other tables as well. To determine which tables to lock and whether a read or write lock is needed, the concept of an immediate and complete family for a table is used:

- An *immediate family* is the set of all tables that reference the table with a FOREIGN KEY reference clause.
- A *complete family* is the immediate family, plus all tables that reference the immediate family tables, and so on.

For the complete family, only one type of delete action is permitted, and a restricted delete (NO ACTION) anywhere in the complete family for a table overrides any cascaded delete conditions.



Warning: *An option to the DELETE statement (OVERRIDE REFCHECK) can be used to omit any checks for referential integrity but should be used only with great caution and a clear understanding of its actions.*

Assume a database has two fact tables named **Fact1** and **Fact2**; three dimension tables named **Product**, **Market**, and **Period**; and two outboard tables named **Brand** and **Monthname**, with references as illustrated by the following figure.

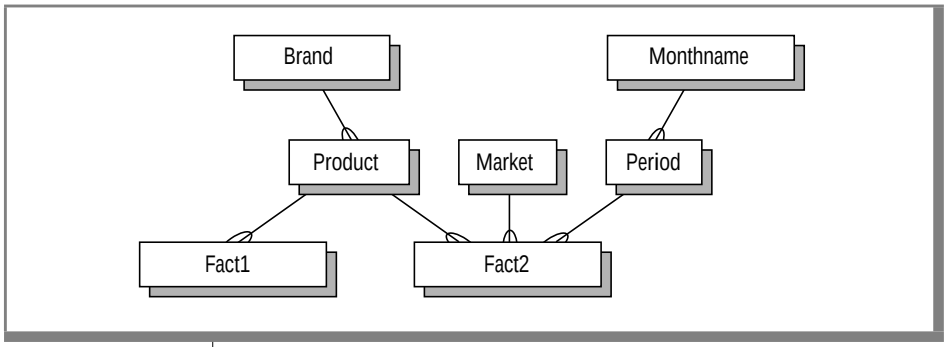


Figure 2-5
Schema Example

The following table defines the relationships among these tables in terms of immediate and complete families for delete locks.

Table Name	Immediate Family	Complete Family
Fact1	None	None
Fact2	None	None
Product	Fact1, Fact2	Fact1, Fact2
Market	Fact2	Fact2
Period	Fact2	Fact2
Brand	Product	Product, Fact1, Fact2
Monthname	Period	Period, Fact2

A delete operation (and the FOR DELETE option to the LOCK TABLE statement) uses a special delete lock that provides necessary and sufficient access to all involved tables. A delete lock locks the named table for write (exclusive) access. The delete lock also locks all tables in the immediate family for either read or write access, depending on the referential action specified when the tables were created. It then locks all tables in the immediate family of these tables in a similar manner. The complete family is locked for write access only if all ON DELETE actions are CASCADE. Otherwise, only the tables in the immediate family are locked for read access.

The following examples illustrate how tables are locked and referential integrity is preserved during delete operations.

Example

This example illustrates a delete operation in which all table references are defined for cascaded deletes.

Assume the **Brand** table is referenced from the **Product** table as follows:

```
brandkey char(3) not null,
...
foreign key brandkey references brand (brandkey)
   on delete cascade
```

Assume the **Product** table is referenced from the **Fact** table as follows:

```
prodkey char(5) not null,
...
foreign key prodkey references product (prodkey)
   on delete cascade
```

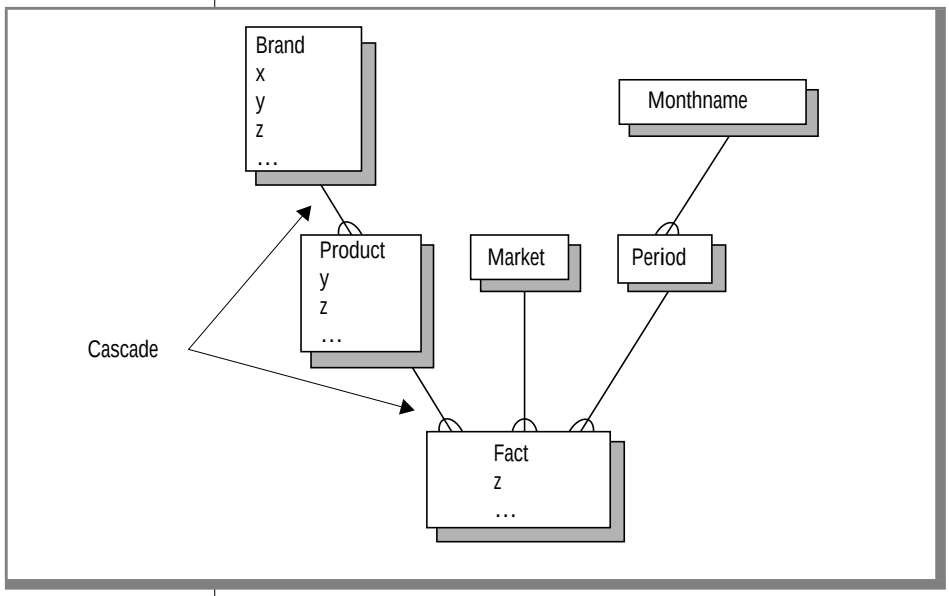


Figure 2-6
Delete Cascade

If a row is to be deleted from the **Brand** table, any row in **Product** that references (has a foreign key value that matches the primary key of) the deleted row in **Brand** is also deleted. Any row in **Fact** that references a deleted row in **Product** is also deleted. The following table illustrates how cascaded deletes work in this case.

From Brand, delete row containing:	Rows deleted from:		
	Brand	Product	Fact
x	Yes	Not present	Not present
y	Yes	Yes	Not present
z	Yes	Yes	Yes

When the delete lock is applied to the **Brand** table for this operation, **Brand** and all tables in its complete family are locked for write access because rows might be deleted from any of those tables. This lock denies access by other users to any tables in the complete family until the delete lock is released.

Example

This example illustrates the effect of a restricted (NO ACTION) delete on a family of tables. Delete operations are treated as if all references were restricted.

Assume the **Monthname** table is referenced from the **Period** table as follows:

```
monkey char(3) not null,
...
foreign key monkey references monthname (monkey)
   on delete no action
```

The type of reference from **Fact** to **Period** does not affect operations on **Monthname** because the **Period**-to-**Monthname** reference is NO ACTION. Nothing can be deleted from **Period**.

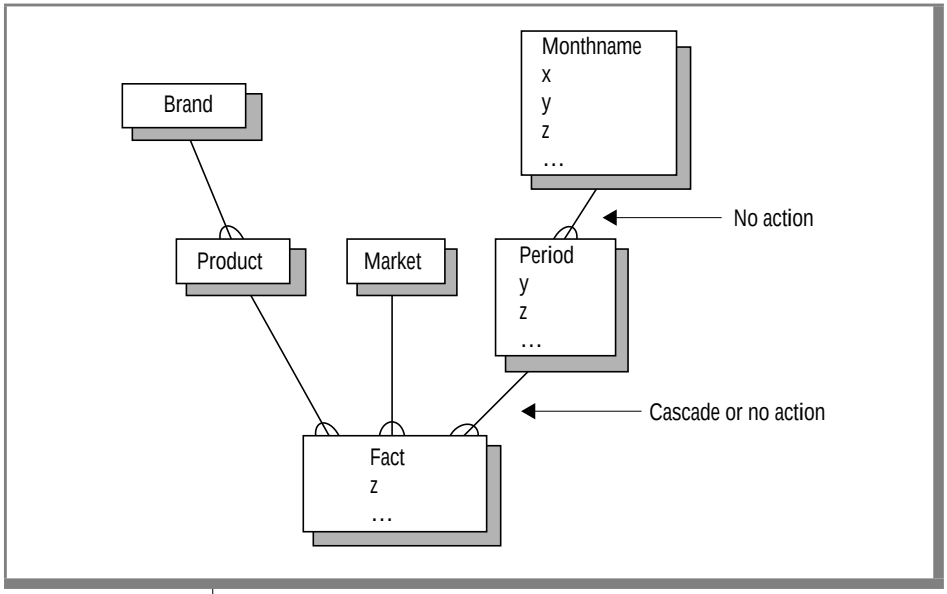


Figure 2-7
Delete No Action

If a row to be deleted from the **Monthname** table is referenced by a row in **Period**, the row is not deleted from **Monthname**. If it is not referenced by a row in **Period**, it is deleted. Because no rows can be deleted from **Period**, it is not necessary to access the **Fact** table to check for referencing rows. The following table illustrates how restricted deletes work in this case.

From Monthname, delete row containing:	Rows deleted from:		
	Monthname	**Period**	**Fact**
x	Yes	No rows deleted from **Period**	No rows deleted from **Fact**
y	No		
z	No		

When the delete lock is applied to **Monthname** for this operation, the **Monthname** table is locked for write access, and **Period**—the only table in its immediate family—is locked for read access. No lock is applied to **Fact**.

Example

This example illustrates a delete operation with a restricted delete not in the immediate family but somewhere in the complete family, which includes both restricted and cascaded references. Delete operations are treated as if all references were restricted.

Assume the **Brand** table is referenced from the **Product** table as follows:

```
brandkey char(3) not null,
...
foreign key brandkey references brand (brandkey)
   on delete cascade
```

Assume the **Product** table is referenced from the **Fact1** table with a cascaded delete as follows:

```
prodkey char(5) not null,
...
foreign key prodkey references product (prodkey)
   on delete cascade
```

Assume the **Product** table is referenced from the **Fact2** table with a restricted delete as follows:

```
prodkey char(5) not null,
...
foreign key prodkey references product (prodkey)
   on delete no action
```

Even though the reference from **Product** to **Brand** is cascade, because another reference in the complete family is restricted, it is just as if **Brand** were referenced from **Product** with a restricted delete.

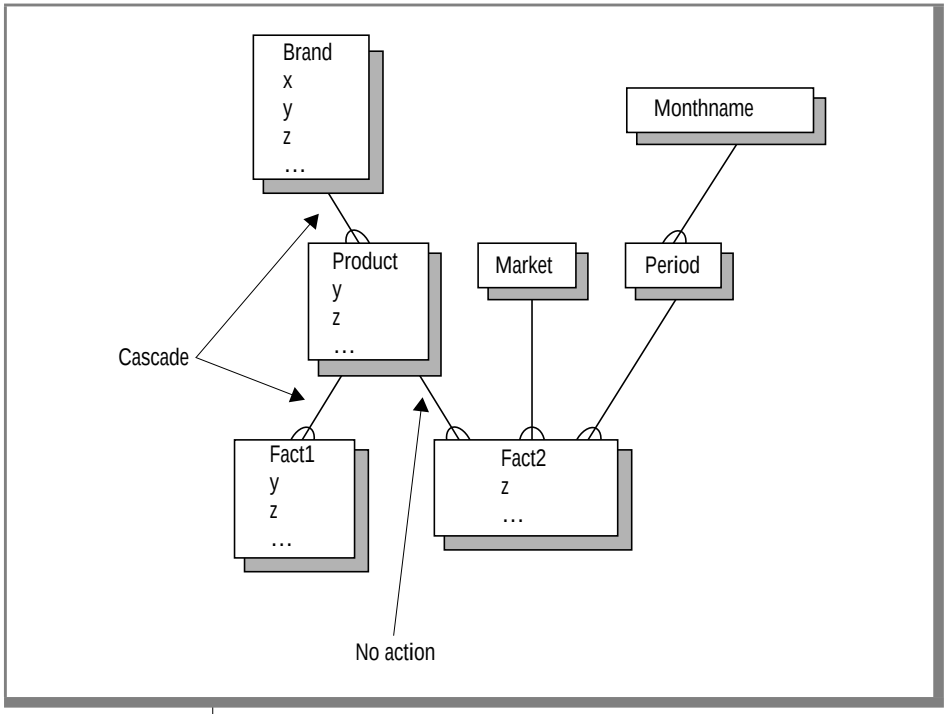


Figure 2-8
Restricted Delete

If a row to be deleted from **Brand** is referenced by **Product**, that row in **Brand** is not deleted. The following table illustrates how the combination of restricted and cascaded deletes works in this case.

From Brand, delete row containing:	Rows deleted from:			
	Brand	**Product**	**Fact1**	**Fact2**
x	Yes	Not present	No rows deleted from **Fact1**	No rows deleted from **Fact2**
y	No	No		
z	No	No		

This example does not indicate what happens in **Fact2** when a row is to be deleted from **Market**, **Period**, or **Monthname**. The relationships defined for the complete families of each of those tables determine the behavior in those families. For example, if the **Market** table is referenced by **Fact2** with a cascaded delete, a row deleted from the **Market** table can cause any corresponding referencing rows to be deleted from **Fact2**. The NO ACTION reference between **Product** and **Fact2** does not affect references between **Fact2** and other tables.

When the delete lock is applied to **Brand** for this operation, **Brand** is locked for write access. **Product**, the only table in its immediate family, is locked for read access. The **Fact1** and **Fact2** tables are not locked. Even though y is not referenced by **Fact2**, it is not deleted from **Brand**, **Product**, or **Fact1**.

For more examples that illustrate how the ON DELETE clause and DELETE FROM statement affect referential integrity, refer to the DELETE statement in the *SQL Reference Guide*.

Chapter

3

Schema Design

Transaction Processing Versus Decision Support 3-3
Transaction-Processing Databases 3-4
Decision-Support Databases 3-5

Star Schemas 3-6
Performance of Star Schemas 3-8
Terminology 3-8
Simple Star Schemas 3-8
Multiple Fact Tables 3-10
Multicolumn Foreign Key 3-12
Outboard Tables 3-13
Multistar Schemas 3-14
Views 3-17

Considerations for Schema Design 3-18
Schema Building Blocks 3-20
Example: Salad Dressing Database 3-23

Analyzing Your Schema 3-24
Browsing the Dimension Tables 3-24
Querying the Fact Table 3-25
Determining Which Attributes to Include 3-25

Schema Examples 3-27
Reservation System Database 3-27
Investment Database 3-29
Health Insurance Database 3-31

In This Chapter

Schema design greatly influences both database performance and the ease with which users retrieve information. This chapter assumes that you are familiar with relational databases and includes the following sections:

- Transaction Processing Versus Decision Support
- Star Schemas
- Considerations for Schema Design
- Analyzing Your Schema
- Schema Examples

Transaction Processing Versus Decision Support

In theory, the relational model supports databases for both transaction processing and decision support. In reality, compromises must be made in the design of database management software to optimize often- conflicting design objectives. Transaction-processing databases are optimized for the insert, update, and delete operations used to capture data, whereas decision-support databases are optimized for query operations used to analyze the data. Data for decision-support systems is often captured by online transaction-processing systems and then loaded into a decision-support system.

Transaction-Processing Databases

Transaction-processing systems are designed to capture information and to be updated quickly. They are constantly changing and are often online 24 hours a day. Examples of transaction-processing systems include order entry systems, scanner-based point-of-sale registers, automatic teller machines, and airline reservation applications. These systems provide operational support to a business and are used to run a business.

Transaction-processing systems have the following characteristics:

- High transaction rate

  To ensure high throughput, transactions are simple and touch as few tables as possible.

- Constant change

  Transactions occur in large numbers, and their changes are largely uncontrolled and unpredictable, within the limits of the system.

- Join paths

  Join paths can be random and cyclic, and they are interpreted at the time of the query.

- No redundancy

  Redundant and aggregate data is avoided in order to ensure data integrity and reduce lockout contention.

- Relational integrity

  The reliability of the data depends on transaction integrity. Relational integrity checks are too slow and require much structural complexity.

- Predictable SQL queries

  To ensure consistent response time, SQL statements are simple, predefined, and carefully tested. Indexes are optimized for these statements but are otherwise avoided because they adversely affect update and insert performance.

- Recoverability

  To ensure against data loss, two-phase commit and rollback mechanisms, continuous transaction logs, and mirrored disk technology are employed.

These goals are achieved by database schemas with a high degree of normalization—schemas that contain large numbers of tables connected by complex join paths. Normalization provides fast transaction response time and a complex schema that is easily manipulated by the applications that use it, but difficult to understand by the people who need the data.

Decision-Support Databases

Decision-support systems are designed to allow analysts to extract information quickly and easily. The data being analyzed is often historical: daily, weekly, and yearly results. Examples of decision-support systems include applications for analysis of sales revenue, marketing information, insurance claims, and catalog sales. A decision-support database within a single business can include data from beginning to end: from receipt of raw material at the manufacturing site, entering orders, tracking invoices, and monitoring database inventory to final consumer purchase. These systems are used to manage a business. They provide the information needed for business analysis and planning.

Decision-support systems have the following characteristics:

- Understandability

  Data structures must be readily understood by users, often requiring denormalization and precomputed aggregations (summary data).

- Relatively infrequent changes

  Most changes to the database occur in a controlled manner when data is loaded at regular intervals.

- Join paths

  Join paths are simple, noncyclic, and based on business relations. They are defined when the database is built.

- Relational integrity

  Relational integrity, necessary to ensure correct results, is built into the database when the data is loaded or deleted.

- Unpredictable and complex SQL queries

  SQL query statements submitted against the database vary considerably and unpredictably from query to query. They can contain long, complex SELECT statements that make comparisons or require sequential processing. These queries might reference many thousands, millions, or even billions of records in a database.

- Large result sets

  Extensive and frequent browsing must be supported.

- Recoverability

  Regular backups, or snapshots, of the static database ensure against data loss.

Red Brick Decision Server supports all types of schemas, but the goals of a decision-support system are often achieved by database schemas referred to as *star schemas*, which are simple in structure with relatively few tables and well-defined join paths. This structure, in contrast to the normalized structure used for transaction-processing databases, provides fast query response time and a simple schema that is easily understood by the analysts who use it, even those who are not familiar with database structures.

The following sections describe two types of star schemas, a simple star schema and a multistar schema.

Star Schemas

A star schema consists of fact tables and dimension tables. *Fact tables* contain the quantitative or factual data about a business—the information being queried. This information is often numerical, additive measurements and can consist of many columns and millions or billions of rows. *Dimension tables* are usually smaller and hold descriptive data that reflects the dimensions, or attributes, of a business. SQL queries then use joins between fact and dimension tables and constraints on the data to return selected information.

Fact and dimension tables differ from each other only in their use within a schema. Their physical structure and the SQL syntax used to create the tables are the same. In a complex schema, a given table can act as a fact table under some conditions and as a dimension table under others. The way in which a table is referred to in a query determines whether a table behaves as a fact table or a dimension table.

Even though they are physically the same type of table, it is important to understand the difference between fact and dimension tables from a logical point of view. To demonstrate the difference between fact and dimension tables, consider how an analyst looks at business performance:

- A salesperson analyzes revenue by customer, product, market, and time period.
- A financial analyst tracks actuals and budgets by line item, product, and time period.
- A marketing person reviews shipments by product, market, and time period.

The facts—what is being analyzed in each case—are revenue, actuals and budgets, and shipments. These items belong in fact tables. The business dimensions—the *by* items—are product, market, time period, and line item. These items belong in dimension tables.

For example, a fact table in a sales database, implemented with a star schema, might contain the sales revenue for the products of the company from each customer in each geographic market over a period of time. The dimension tables in this database define the customers, products, markets, and time periods used in the fact table.

A well-designed schema provides dimension tables that allow a user to browse a database to become familiar with the information in it and then to write queries with constraints so that only the information that satisfies those constraints is returned from the database.

Performance of Star Schemas

Performance is an important consideration of any schema, particularly with a decision-support system in which you routinely query large amounts of data. Red Brick Decision Server supports all schema designs. However, star schemas tend to perform the best in decision-support applications.

Terminology

The terms *fact table* and *dimension table* represent the roles these objects play in the logical schema. In terms of the physical database, a fact table is a *referencing* table. That is, it has foreign key references to other tables. A dimension table is a *referenced* table. That is, it has a primary key that is a foreign key reference from one or more tables.

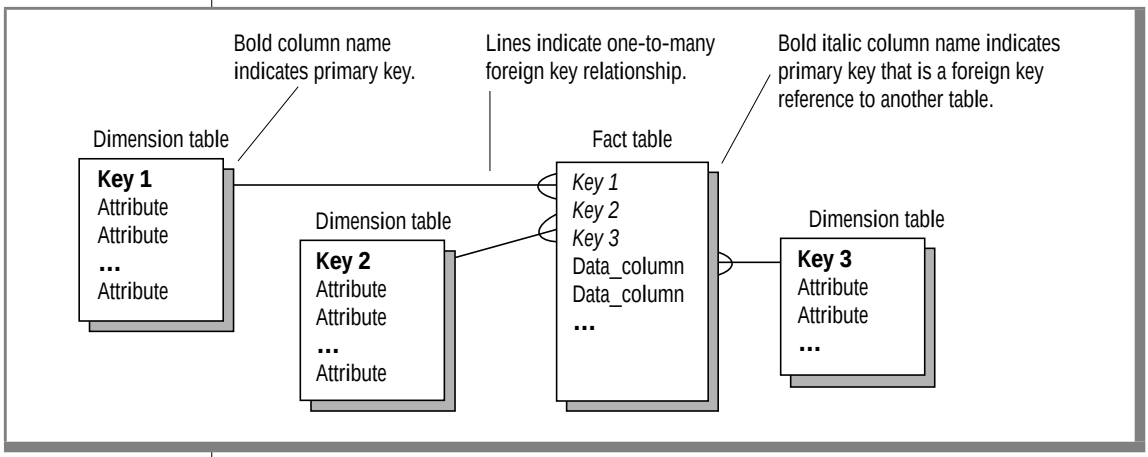
Simple Star Schemas

Any table that references or is referenced by another table must have a *primary key*, which is a column or group of columns whose contents uniquely identify each row. In a simple star schema, the primary key for the fact table consists of one or more *foreign keys*. A foreign key is a column or group of columns in one table whose values are defined by the primary key in another table. In Red Brick Decision Server, you can use these foreign keys and the primary keys in the tables that they reference to build STAR indexes, which improve data retrieval performance.

When a database is created, the SQL statements used to create the tables must designate the columns that are to form the primary and foreign keys.

The following figure illustrates the relationship of the fact and dimension tables within a simple star schema with a single fact table and three dimension tables. The fact table has a primary key composed of three foreign keys, **Key1**, **Key2**, and **Key3**, each of which is the primary key in a dimension table. Nonkey columns in a fact table are referred to as data columns. In a dimension table, they are referred to as attributes.

Figure 3-1
Simple Star Schema



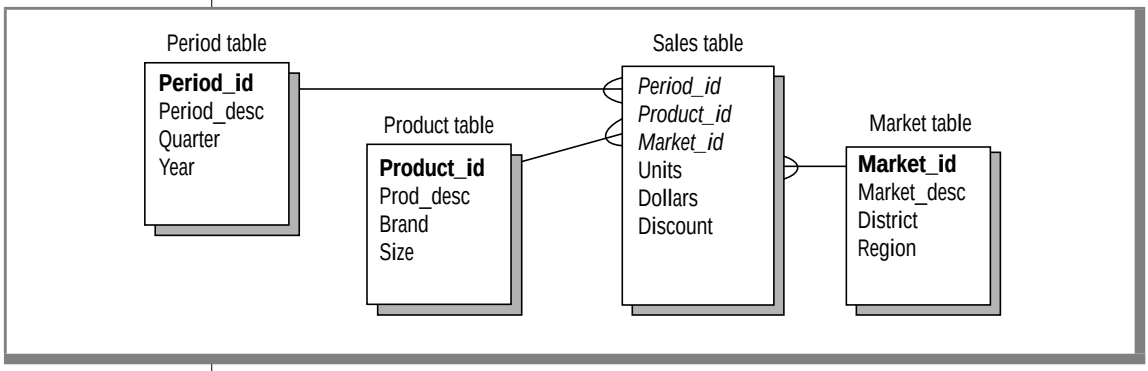
In the figures used to illustrate schemas:

- The items listed within the box under each table name indicate columns in the table.
- Primary key columns are labeled in bold type.
- Foreign key columns are labeled in italic type.
- Columns that are part of the primary key and are also foreign keys are labeled in bold italic type.
- Foreign key relationships are indicated by lines connecting tables.

  Although the primary key value must be unique in each row of a dimension table, that value can occur multiple times in the foreign key in the fact table—a many-to-one relationship.

The following figure illustrates a sales database designed as a simple star schema. In the fact table **Sales**, the primary key is composed of three foreign keys, **Product_id**, **Period_id**, and **Market_id**, each of which references a primary key in a dimension table.

Figure 3-2
Sales Database



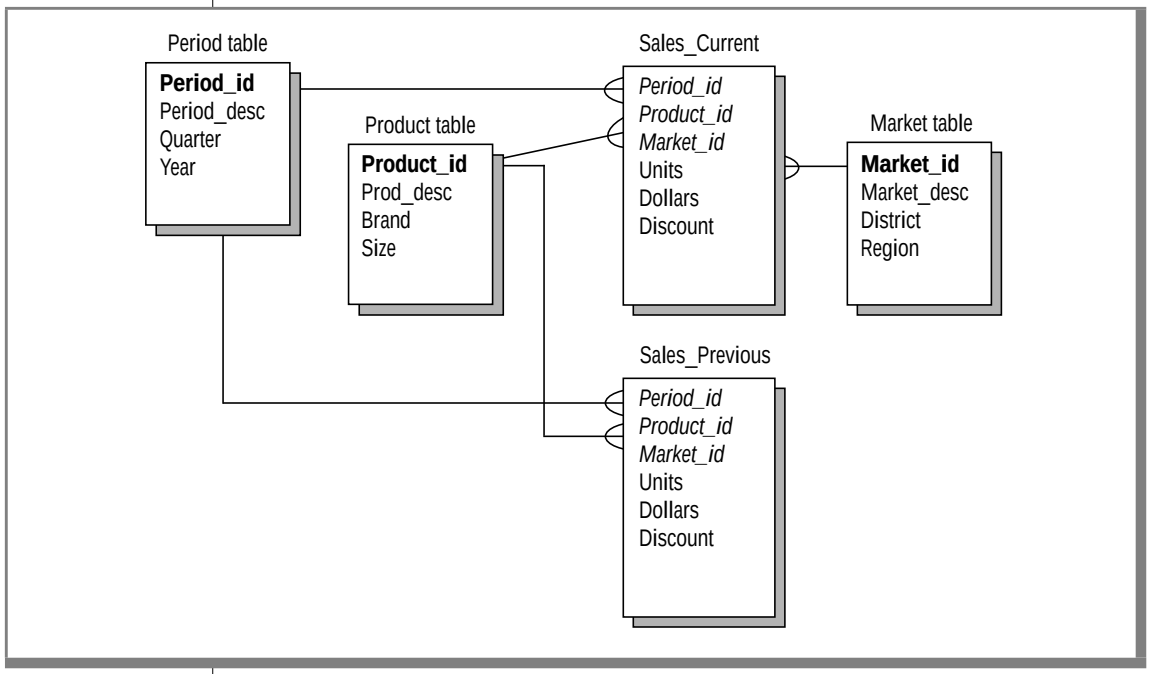
Many-to-one relationships exist between the foreign keys in the fact table and the primary keys they reference in the dimension tables. For example, the **Product** table defines the products. Each row in the table represents a distinct product and has a unique product identifier. That product identifier can occur multiple times in the **Sales** table representing sales of that product during each period and in each market.

Multiple Fact Tables

A star schema can contain multiple fact tables. In some cases, multiple fact tables exist because they contain unrelated facts; for example, invoices and sales. In other cases, they exist because they improve performance. For example, multiple fact tables are often used to hold various levels of aggregated (summary) data, particularly when the amount of aggregation is large; for example, daily sales, monthly sales, and yearly sales.

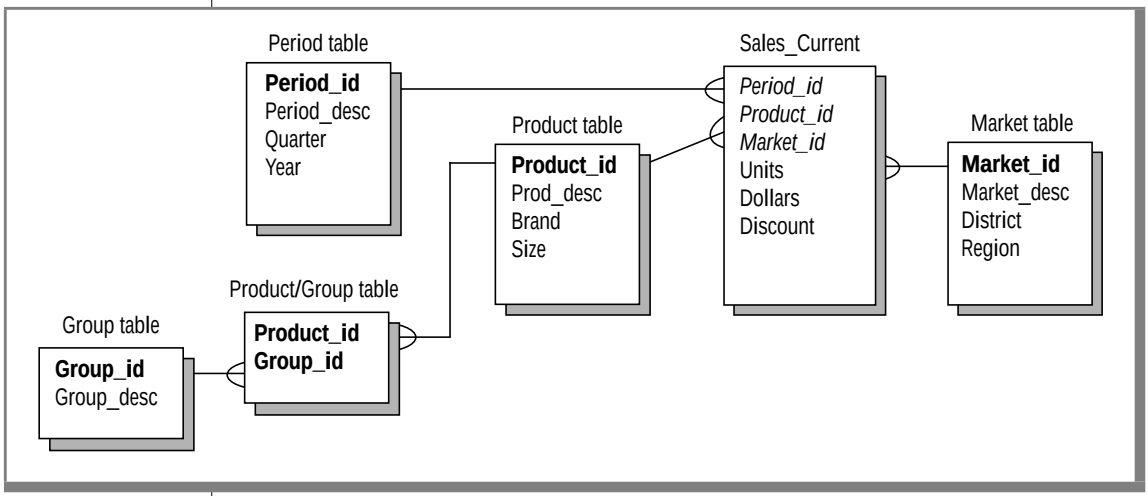
The following figure illustrates the **Sales** database with an additional fact table for sales from the previous year.

Figure 3-3
Sales Database with Additional Dimension



Another use of a referencing table is to define a many-to-many relationship between some dimensions of the business. This type of table is often known as a cross-reference or associative table. For example, in the **Sales** database, each product belongs to one or more groups, and each group contains multiple products, a many-to-many relationship that is modeled by establishing a referencing table that defines the possible combinations of products and groups.

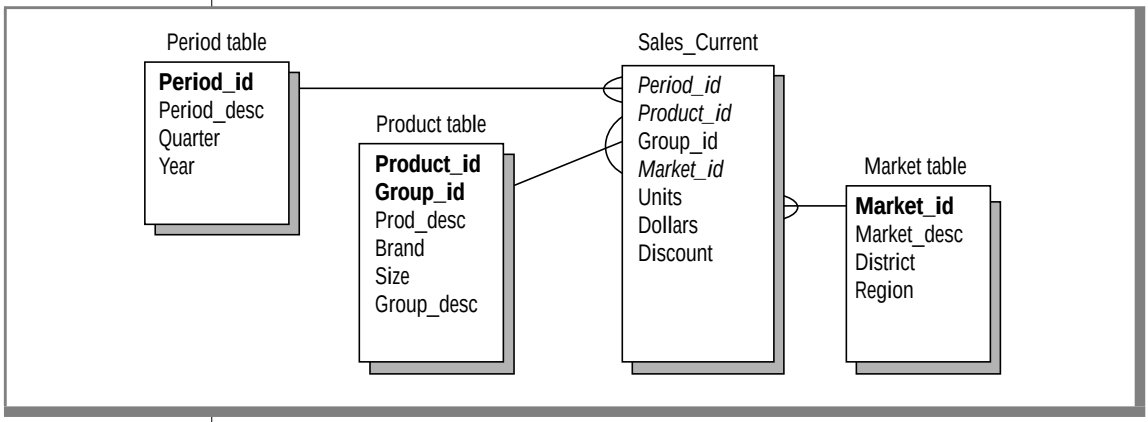
Figure 3-4
Sales Database with Cross-Reference Table



Multicolumn Foreign Key

Another way to define a many-to-many relationship is to have a dimension table with a multicolumn primary key that is a foreign key reference from a fact table. For example, in the **Sales** database, each product belongs to one or more groups, and each group contains multiple products, a many-to-many relationship. This relationship is modeled by defining a multicolumn foreign key in the **Sales_Current** table that references the **Product** table, as in the following example.

Figure 3-5
Sales Database with Multicolumn Foreign Key

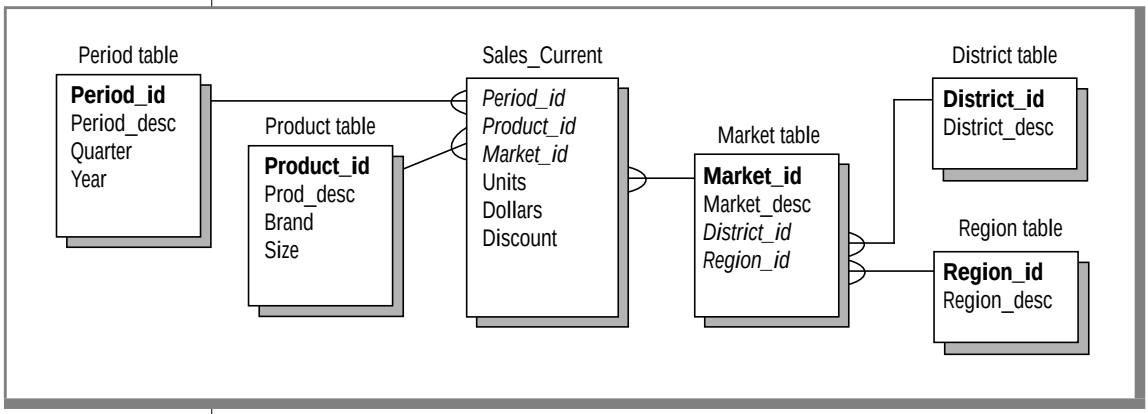


In the preceding figure, the **Product_id** and **Group_id** columns are the two-column primary key of the **Product** table and are a two-column foreign key reference from the **Sales_Current** table.

Outboard Tables

Dimension tables can also contain one or more foreign keys that reference the primary key in another dimension table. The referenced dimension tables are sometimes referred to as *outboard*, *outrigger*, or *secondary* dimension tables. The following figure includes two outboard tables, **District** and **Region**, which define the ID codes used in the **Market** table.

Figure 3-6
Sales Database with Outboard Tables



In the preceding figure, the **Market** table, because it is both a referencing and referenced table, can behave as a fact (referencing) or dimension (referenced) table, depending on how it is used in a query.

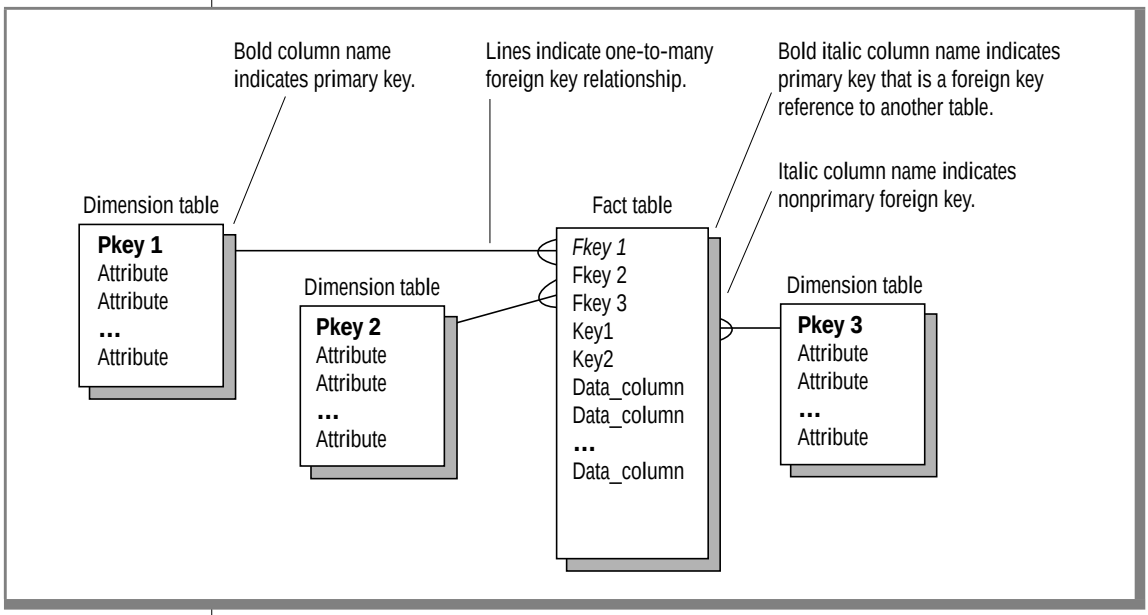
Multistar Schemas

In a simple star schema, the primary key in the fact table is formed by concatenating the foreign key columns. In some applications, however, the concatenated foreign keys might not provide a unique identifier for each row in the fact table. These applications require a multistar schema.

In a *multistar* schema, the fact table has both a set of foreign keys, which reference dimension tables, and a primary key, which consists of one or more columns that provide a unique identifier for each row. The primary key and the foreign keys are not identical in a multistar schema. This fact distinguishes a multistar schema from a single-star schema.

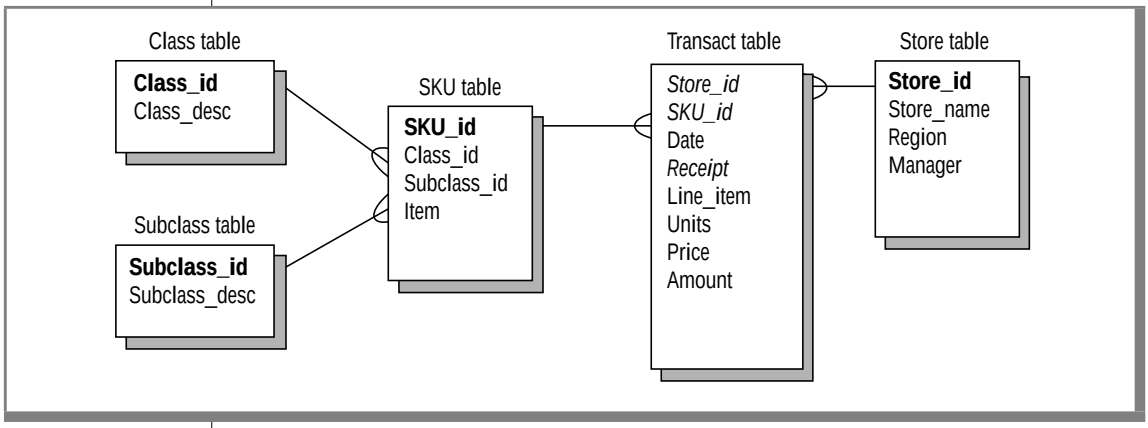
The following figure illustrates the relationship of the fact and dimension tables within a multistar schema. In the fact table, the foreign keys are **Fkey1**, **Fkey2**, and **Fkey3**, each of which is the primary key in a dimension table. Unlike the simple star schema, these columns do not form the primary key in the fact table. Instead, the two columns **Key1** and **Key2**, which do not reference any dimension tables, and **Fkey1**, which does reference a dimension table, are concatenated to form the primary key. The primary key can consist of any combination of foreign key and other columns in a multistar schema.

Figure 3-7
Relationship of Fact and Dimension Tables in Multistar Schema



The following figure illustrates a retail sales database designed as a multistar schema with two outboard tables. The fact table **Transact** records daily sales in a rolling seven-day database. The primary key for the fact table consists of three columns: **Date**, **Receipt**, and **Line_item**. These keys together provide the unique identifier for each row. The foreign keys are the columns for **Store_id** and **SKU_id**, which reference the **Store** and **SKU** (storekeeping unit) dimension tables. Two outboard tables, **Class** and **Subclass**, are referenced by the **SKU** dimension table.

Figure 3-8
Multistar Schema with Two Outboard Tables



In this database schema, analysts can query the transaction table to obtain information on sales of each item, sales by store or region, sales by date, or other interesting information.

In a multistar schema, unlike a simple star schema, the same value for the concatenated foreign key in the fact table can occur in multiple rows, so the concatenated foreign key no longer uniquely identifies each row. For example, in this case the same store (**Store_id**) might have multiple sales of the same item (**SKU_id**) on the same day (**Date**). Instead, row identification is based on the primary key or keys. Each row is uniquely identified by **Date**, **Receipt**, and **Line_item**.

Views

In some databases, schema design can be simplified by the use of views, which effectively create a virtual table by selecting a combination of rows and columns from an existing table or combination of tables. For example, a view that selects employee names and telephone extensions from an employee database produces a company phone list but does not include confidential information such as addresses and salaries. A view that selects transactions that occur within a given time period avoids the need to constrain queries to that time period.

Views are useful for a wide variety of purposes, including the following:

- Increasing security
- Simplifying complex tables to give users a view of only what they need
- Simplifying query constraints
- Simplifying administrative tasks, such as granting table authorizations
- Hiding administrative changes to users

  The database schema changes design, but the view to the user remains the same.

A view is created with a CREATE VIEW statement.

Additionally, you can create precomputed views so that queries are automatically rewritten to access the appropriate aggregate table. For information on precomputed views and automatic query rewriting, refer to the *Informix Vista User's Guide*.

Considerations for Schema Design

The schema design for a database affects its usability and performance in many ways, so it is important to make the initial investment in time and research to design a database that meets the needs of its users. This section is not intended to provide a detailed guide to database design, but only to present some ideas to consider in designing a database.

A well-designed schema takes into account the following considerations:

- What are the processes of the business?

  Identify the main processes of the business; for example, taking orders for the product, filling out insurance claims, or tracking promotions. These processes are different for every business, but they must be clearly identified and defined in order to create a useful database. The people who know the processes are the people who work in the business, and interviews are essential to determine these processes.

- What do the users want to accomplish with the database?

  The database should reflect the business, both in what it measures and tracks and in the terminology used to describe the facts and dimensions of the business. Interviews with managers and users will reveal what they want to know, how they measure the business, what criteria they use to make decisions, and what words they use to describe these things. This information helps determine the contents of the fact and dimension tables.

- Where will the data come from?

  The data to populate the tables in the database must be complete enough to be useful and must be valid, consistent data. An analysis of the proposed input data and its sources will reveal whether the available data can support the proposed schema.

- What are the dimensions of the business and their attributes that will be reflected by the dimension tables?

  Independent dimensions should be represented by separate tables. If dimensions are not independent, they can be combined in a single table. Attributes are usually textual and discrete values; for example, product descriptions or geographic locations. They are used to form query constraints and to determine report breaks. The interviews and data analysis will provide guidance in setting up these tables.

- Are the dimensions going to change over time?

  If a dimension changes frequently, it probably should be measured as a fact, not stored as a dimension.

- What facts should be measured?

  Facts are usually numerical and continuous values; for example, revenue or inventory. Facts that are additive can be summed to produce valid measures in reports. For example, sales for each month are additive and can be summed to produce year-to-date totals. Month-end inventory balances, however, are not additive in the sense that a yearly total of month-end inventory balances is of dubious value, but a monthly average might be meaningful.

- Is a family of fact tables needed?

  Facts that are measured with different dimensions or use different timing should be stored in separate tables. For example, a single database can be used for orders, shipments, and manufacturing. Although the facts measured in each area of the business are different, they share some but not all of the same dimensions.

- What is the granularity of the facts?

  Granularity refers to the level of detail of the information stored in each row of the fact table. Each row should hold the same type of data. For example, each row could contain daily sales by store by product or daily line items by store.

  Differing data granularities can be handled by using multiple fact tables (daily, monthly, and yearly tables) or by modifying a single table so that a *granularity flag* (a column to indicate whether the data is a daily, monthly, or yearly amount) can be stored along with the data. Also consider the amounts of data, space, and performance requirements in deciding how to handle different granularities.

- How will changes be handled, and how important is historical information?

  If change occurs infrequently and if historical information is not very important, dimension tables can be modified to reflect only the new reality without any loss of useful data. However, if previous history is important, dimension tables can be modified to reflect both the old and new conditions. If a dimension changes frequently, perhaps it should be considered time dependent and include a time-based attribute; for example, month, quarter, or year.

Schema Building Blocks

The following figures illustrate some common schema examples. Tables named **Fact** or **Fact*x*** represent fact (referencing) tables. The other tables represent dimension (referenced) tables. The following figures apply to both single-star and multistar schemas.

A schema can consist of a single dimension table.

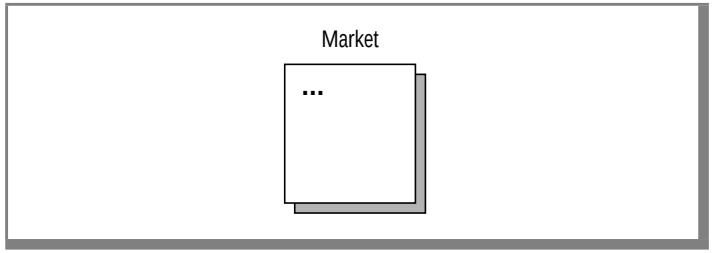


Figure 3-9 Single Dimension Table

A schema can be a star schema with one fact table and one dimension table.

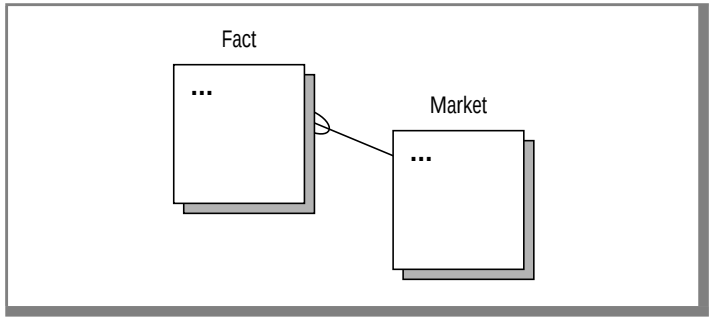


Figure 3-10 Star Schema with One Dimension Table

A schema can be a star schema with one fact table and several dimension tables.

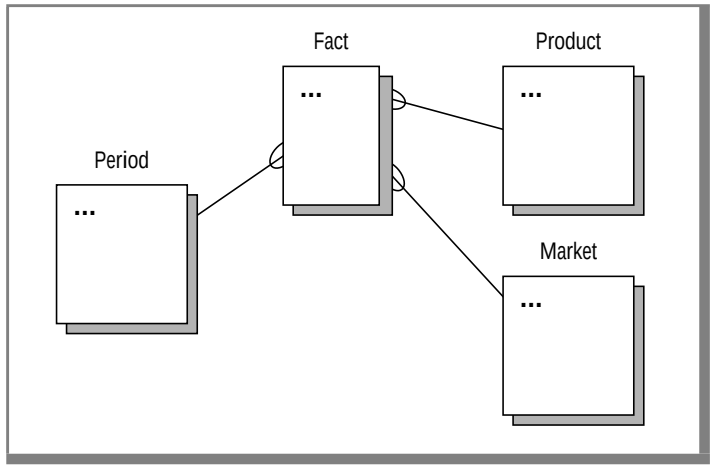


Figure 3-11 Star Schema with Several Dimension Tables

A schema can be a multiple star schema, with a family of fact tables that share some, but not necessarily all, dimension tables.

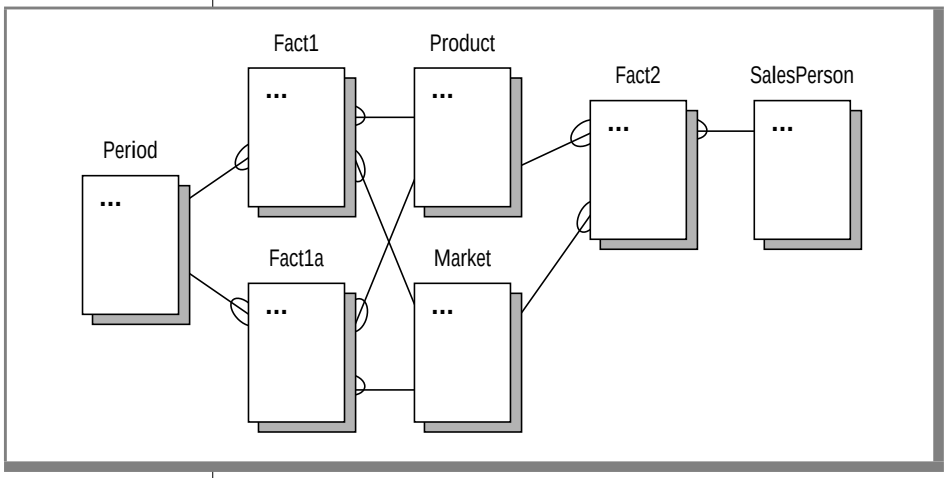


Figure 3-12 Star Schema with Several Fact Tables

A schema can be an extended star schema with dimension tables that reference other dimension tables (outboard tables).

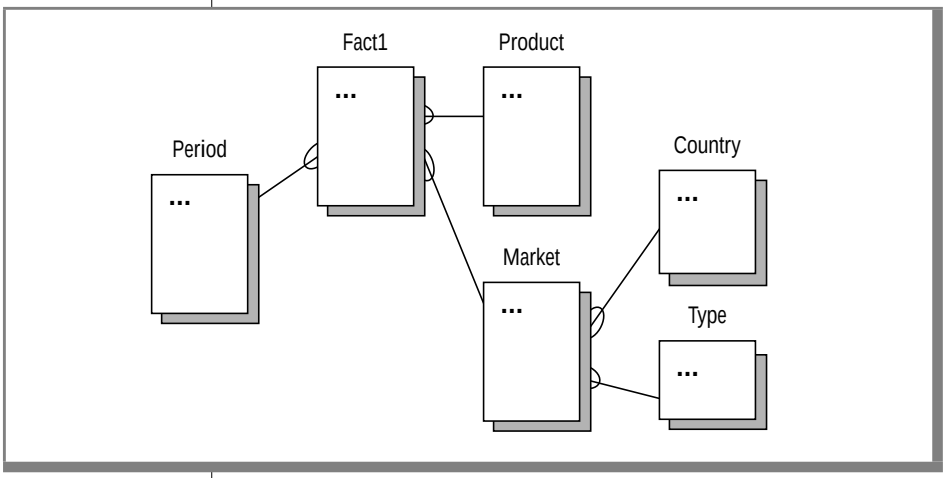


Figure 3-13 Star Schema with Outboard Tables

A schema can be a star schema with a fact table that contains multiple foreign keys that reference single dimension tables.

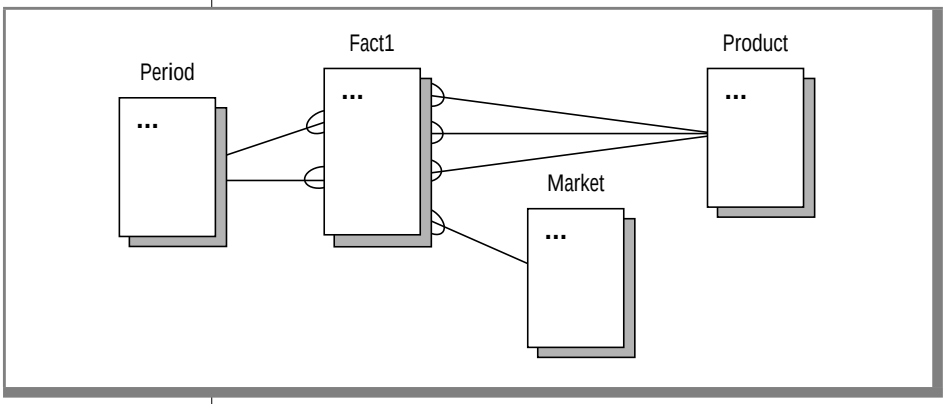


Figure 3-14 Star Schema with Multiple Foreign Keys

Example: Salad Dressing Database

This example illustrates how the schema design affects both usability and usefulness of the database.

This database tracks the sales of salad dressing products in supermarkets at weekly intervals over a four-year period and is a typical consumer-goods marketing database. The salad dressing product category contains 14,000 items at the universal product code (UPC) level. Data is summarized for each of 120 geographic areas (markets) in the U.S. and for each of 208 weekly time periods spanning four years.

The salad dressing database has one fact table, **Sales**, and three dimension tables: **Product**, **Week**, and **Market**, as illustrated in the following figure.

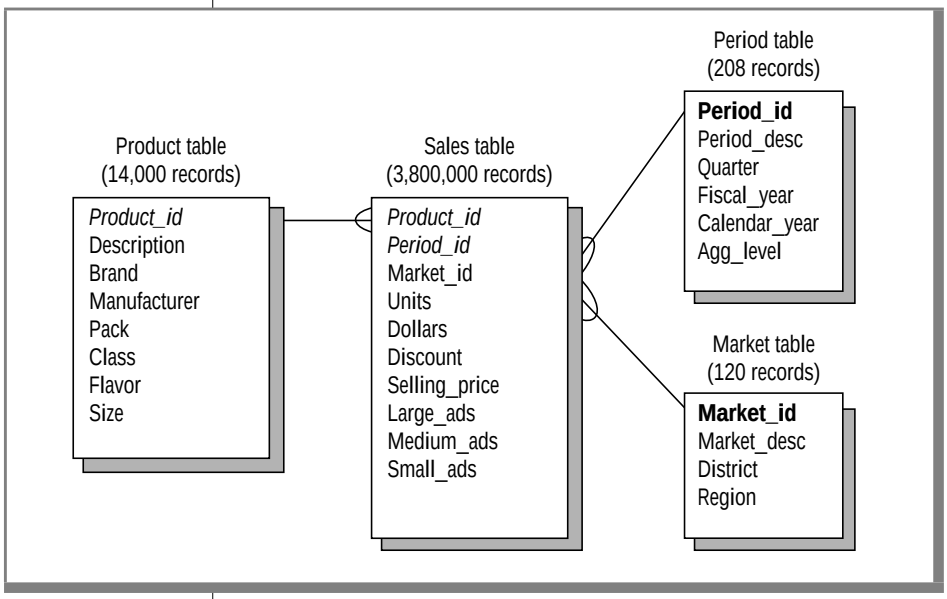


Figure 3-15 Salad Dressing Database Example

Each record in the **Sales** fact table contains a field for each of the three dimensions: **Product**, **Period**, and **Market**. The columns in the **Sales** table containing these fields are the foreign keys whose concatenated values give each row in the **Sales** table a unique identifier. **Sales** also contains seven additional fields that contain values for measures of interest to market analysts.

Each dimension table describes a business dimension and contains one primary key and some attribute columns for that dimension.

Analyzing Your Schema

More often than not, database administrators are presented with an existing database rather than designing one from scratch. To analyze and improve an existing schema, consider the range of values of dimension tables, write queries to perform joins between fact and dimension tables, and determine which attributes to include.

Browsing the Dimension Tables

A convenient way to find the range of values for a specific dimension is to query the dimension table for that dimension. For example, to see what the markets are for the sales data, enter the following statement:

```
select market_desc from market;
```

It displays a list of all the markets, 120 in this case. Similar queries on the **Product** and **Week** tables provide the user with lists of the products and periods covered in the **Sales** table.

Wildcard expressions can be used to narrow the browse list to items that approximate those of interest. For example, if the user is interested in ranch-style dressings, a wildcard expression incorporating `ranch` in the SELECT statement limits the browse list from the **Product** table to those products with `ranch` in the product description instead of 14,000 items.

Browsing through the dimension tables is quicker than issuing a SELECT DISTINCT statement on a fact table, especially if the fact table contains millions of rows of data. Having tables of data that define each dimension of the star schema makes this browsing activity possible. Users can browse the dimensions of the database using the dimension tables to become familiar with the data contents.

Querying the Fact Table

After creating browse lists to determine which markets, products, and time periods are covered in the database, the user looks over these lists to find the markets, products, and time periods of interest. The browse lists return the exact descriptions and spellings, making it easier to write the query constraints correctly. You can then write queries that perform joins between the fact and dimension tables to link the additive data from the fact table to the descriptive data from the dimension tables.

Determining Which Attributes to Include

Nonkey columns in a dimension table are referred to as attributes. To see how attributes are used, consider the **Product** table for the salad dressing database. It has 14,000 items that are identified by their Universal Product Code (UPC), which provides the primary key (**Product_id**). This identifier allows a user to retrieve a unique row. Usually, however, the user does not want data at the UPC level but is interested in higher-level categories such as brand or manufacturer. Attributes permit commonly accessed subsets of an entire group to be differentiated.

For example, the brand attribute allows the 14,000 salad dressing products to be differentiated by brand so that a user can select only products with a specific brand name. Another attribute allows those same 14,000 products to be differentiated by manufacturer. A user analyzing the salad dressings uses the following attributes to select the diet ranch-type salad dressings in 12-ounce bottles from major manufacturers.

Attribute	Possible Values
Class	diet, regular
Flavor	ranch, bleu cheese, thousand island
Size	12 ounces, 8 ounces
Manufacturer	Great Foods, Major Mills, Crafty Cuisine

A well-designed schema includes attributes that reflect the users' potential areas of interest and attributes that can be used for aggregations as well as for selective constraints and report breaks. In addition to the attributes shown in the previous figure, the **Product** table might be expanded to include a wider range of attributes:

UPC (key field)	Viscosity
Distribution	Flavor
Manufacturer	Size
Brand Group	Special Package
Brand	Promotion
Class	

Other types of tables might have only a few attributes. For example, the period dimension for a financial application might need only three attributes:

- Period ID (key field)
- Beginning date
- Ending date

An aggregation level in the **Period** table can be used to distinguish aggregated data from detail data when each type of data is stored in a single table. However, if aggregate and detail data are stored in a single table, each query against that table must constrain on the aggregation level to avoid double-counting.

Attributes do not need to be hierarchical although hierarchy might be required in some cases, as in a general ledger chart of accounts. Multiple hierarchies can also be represented in a single table. For example, in a table that records information with a geographic base, separate geographic hierarchies —physical, sales organization, customer organization—can be recorded in the same table, and any of these attributes can form the basis for constraints.

An attribute can be defined to permit missing values in cases where an attribute does not apply to a specific item or its value is unknown.

A schema design that contains complete, consistent, and accurate attribute fields helps users write queries that they intuitively understand and reduces the support burden on the organization responsible for database management.

Schema Examples

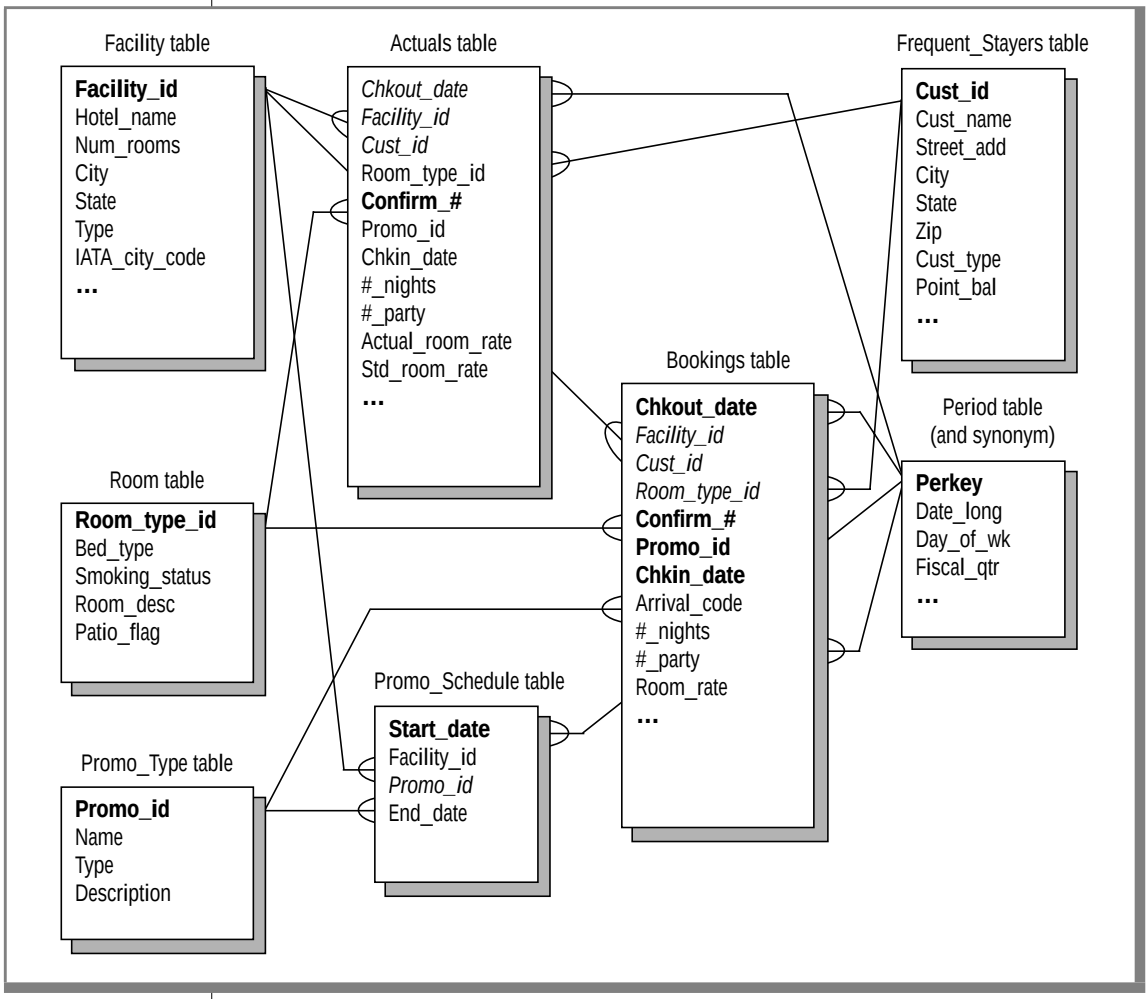
The following examples illustrate common schema designs based on the design considerations that are essential to usability and performance.

Reservation System Database

This example illustrates a multistar schema, in which the primary and foreign keys are not composed of the same set of columns. This design also contains a family of fact tables: a **Bookings** table, an **Actuals** table, and a **Promo_Schedule** table.

This database tracks reservations (bookings) and actual accommodations rented for a chain of hotels, as well as various promotions. It also maintains information about customers, promotions, and each hotel in the chain.

Figure 3-16
Schema for Reservation System



In cases where payment is received in advance (for example, reservation deposits, cable TV subscriptions, automobile insurance), in accordance with proper accounting procedures, transactions must be made to reflect income as it is earned, not when it is received, and the database must be designed to accommodate such transactions.

Investment Database

This example illustrates a schema to handle data aggregations. In this case, daily data is stored in one table and aggregated data (for example, monthly and yearly) in another, rather than combining both levels of aggregation in one table. The ratio of aggregated data to nonaggregated data and knowledge of the expected queries can help determine whether to combine various aggregation levels in a single database or use multiple tables. If aggregated and nonaggregated data is stored in the same table, each query must specify the level of aggregation as a constraint.

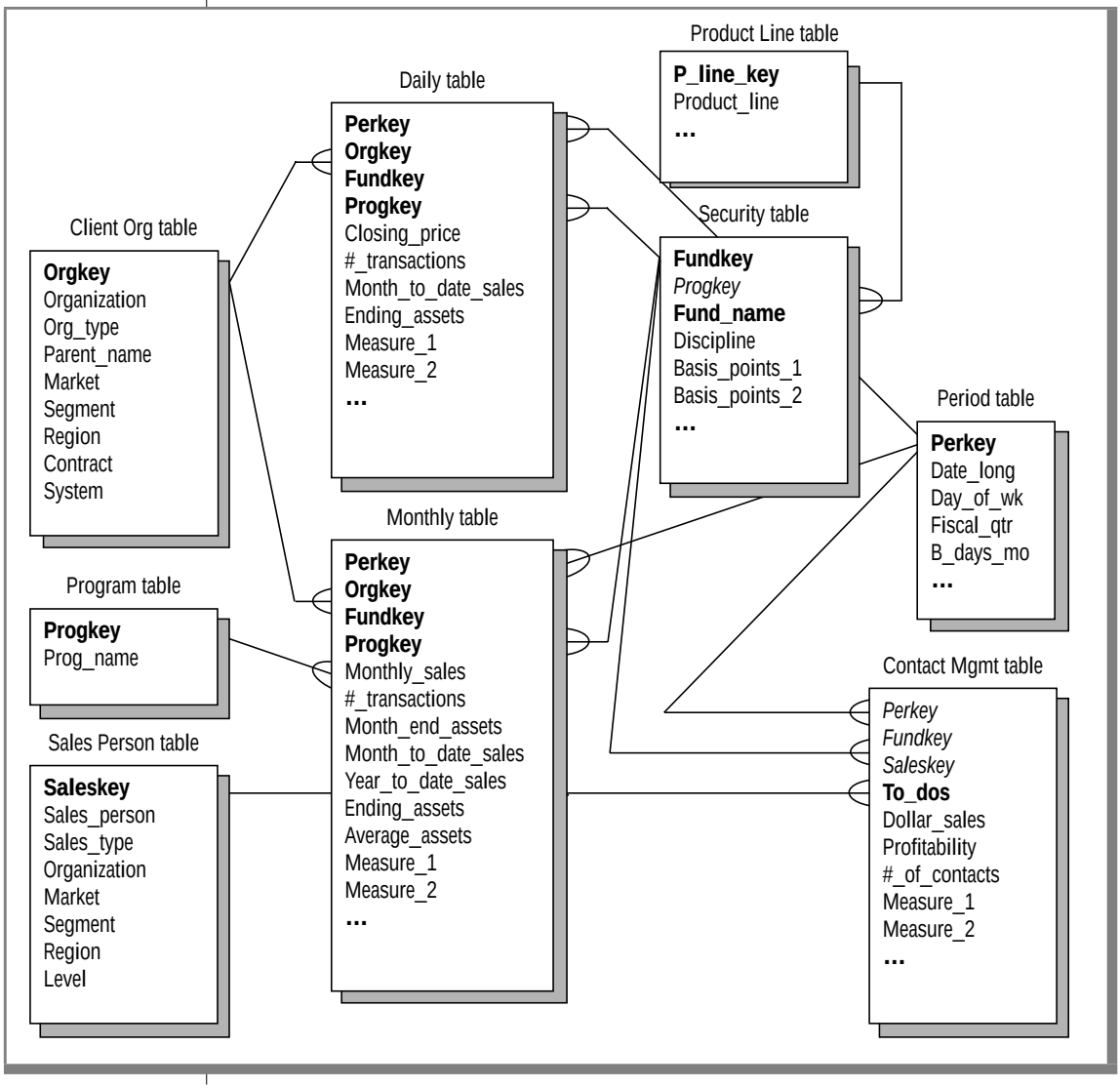
Although separate tables for aggregated data require more disk space, they can sometimes provide performance benefits. Furthermore, the aggregates are usually relatively small compared with the unaggregated fact table. In some situations, this space-for-performance tradeoff is worthwhile.

For example, consider a fact table containing 100 gigabytes of data. If you build three aggregates, the first containing 5 gigabytes of data, the second containing 100 megabytes of data, and the third containing 2 megabytes of data, you are only adding approximately 5 percent of disk space to your fact table. In return, you can have queries that involve a join of a dimension table to the fact table, and they might only need to join the 2-megabyte fact table instead of the 100-gigabyte fact table. Depending on the complexity of the query, it can potentially run thousands of times faster, returning results in seconds rather than minutes, hours, or even days. However, every situation is different, and aggregated data is not the right approach in every case.

With Informix Vista, you can use existing aggregate tables to create precomputed views. The precomputed views allow the system to automatically rewrite queries submitted against a detail table to access an aggregate table. During TMU loads and REORG operations, aggregate tables can be automatically updated based on detail table changes. For detailed information about Vista, refer to the *Informix Vista User's Guide*.

The investment database shown in the following figure tracks sales of investment funds on a daily and monthly basis. It also maintains information about the client organizations, the investment funds, and various trading programs.

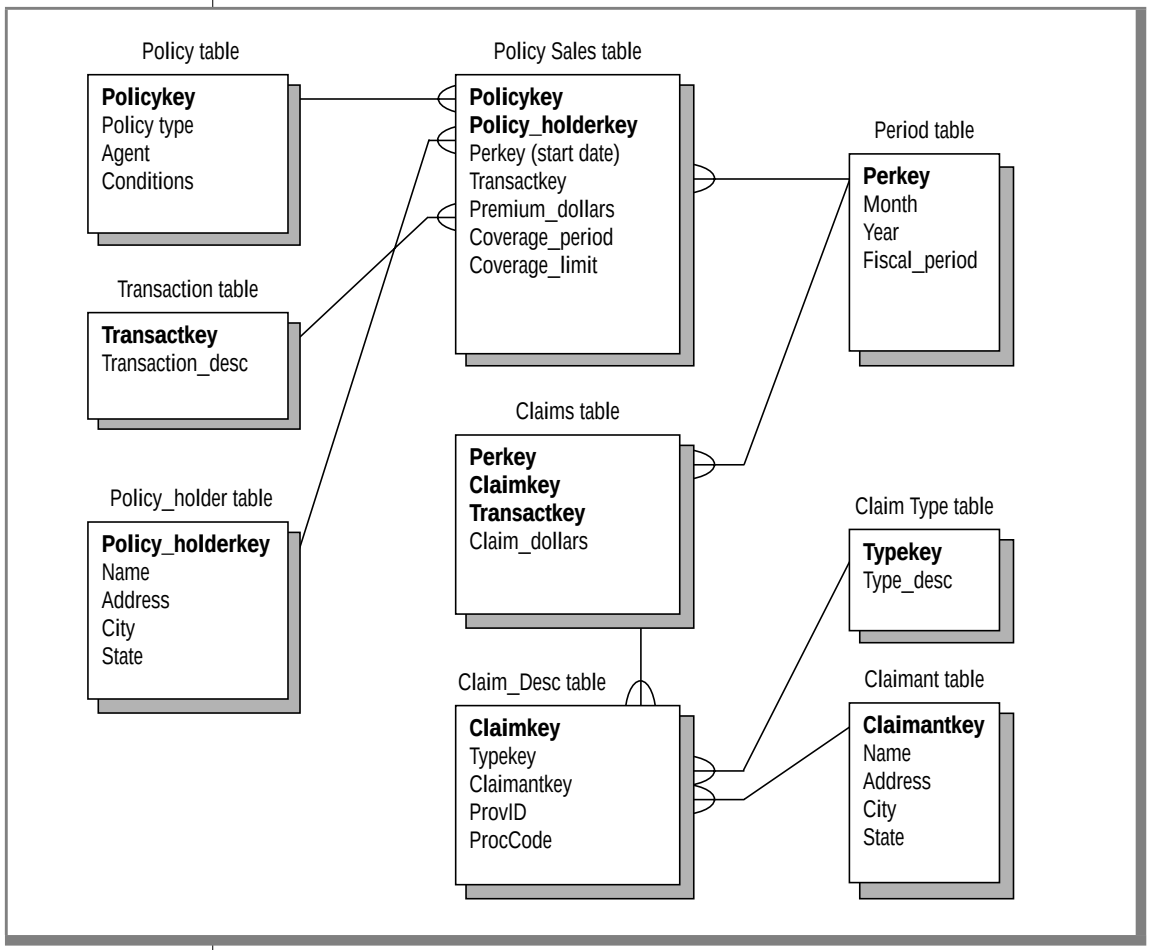
Figure 3-17
Schema for Investment Database



Health Insurance Database

This example illustrates a star schema used for claims analysis by a health care insurance company. This database records policy sales and claims and maintains records of customers, their policies, and claims against those policies.

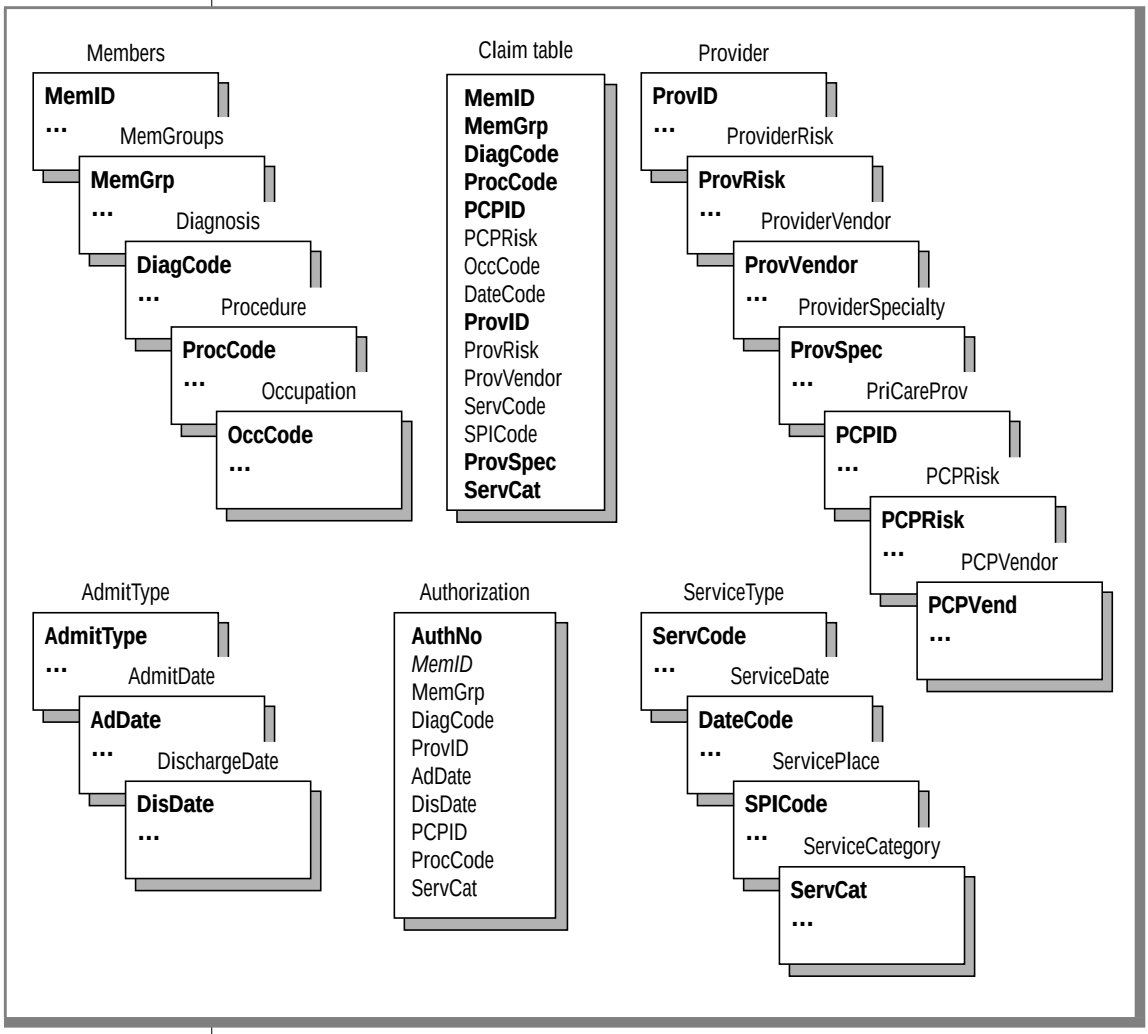
Figure 3-18
Schema for Health Insurance Company



A second example in the health care field illustrates a schema that tracks member claims and authorizations, with many dimensions that include patients and provider information, diagnoses, services performed, and other dimensions of the business. This schema has two fact tables: **Claim**, a star, and **Authorization**, a multistar with a single-column primary key and multiple foreign keys that are not part of the primary key. Any combination of foreign key values can be present multiple times in the Authorization table. The primary key values uniquely identify each row. Both tables have many foreign keys that reference the numerous dimension tables.

In this figure, the primary keys for each table are in boldface, and only the primary keys and foreign keys are labeled. Other attributes within the tables are not shown. Because of the many dimension tables, the many-to-one lines that match the foreign keys in the fact table with the primary keys in the dimension tables are not drawn.

Figure 3-19
Multistar Schema for Health Insurance Company with Multiple Dimension Tables



Chapter

4

Planning a Database Implementation

Organizing Data into Databases 4-3

Determining When to Create Additional Indexes 4-4
STAR Indexes 4-6
B-TREE Indexes 4-9
TARGET Indexes 4-10
Understanding Weakly Selective Constraints 4-10
Choosing the Right Domain Size 4-10
Knowing Your Data 4-11
No Indexes 4-12

Planning for TARGETjoin Processing 4-12
STARjoin Versus TARGETjoin. 4-13
Administration Considerations for TARGETjoin Processing . . . 4-14
Cost Versus Performance 4-14
Load Operations 4-15
Large Dimension Tables 4-16
Multicolumn Foreign Keys 4-17
Parallel TARGETjoin Queries 4-18
TARGET Index DOMAIN Clause 4-18

Planning Disk Storage Organization 4-20
Estimating the Size of User Tables 4-21
Estimating the Size of Indexes. 4-22
Index Fill Factors 4-23
STAR Indexes 4-25
B-TREE Indexes 4-26
TARGET Indexes 4-26

Example: Calculating Table, Index, and System Table Sizes 4-27
Fact1 Table and Its Indexes 4-28
Market Table and Its Indexes 4-29
Product Table and Its Indexes 4-31
Estimating the Size of System Tables 4-32
Size: System Tables 4-33
Total Space for User Tables, Indexes, and System Tables 4-34

Estimating Temporary Space Requirements 4-35
How Optimized Index-Building Operations Use Temporary Space 4-36
Estimating Temporary Space Values for Index-Building Operations 4-37
DIRECTORY Location Values 4-37
THRESHOLD Value 4-38
MAXSPILLSIZE Value 4-39
Online Index-Building Operations 4-39
Offline Index-Building Operations 4-40
Temporary Space Requirements for TARGET Indexes 4-42
How Query Operations Use Temporary Space 4-43
Estimating a QUERY_MEMORY_LIMIT Value for Queries 4-43
Estimating a MAXSPILLSIZE Value for Queries 4-44

Planning for Segmented Storage 4-45
Determining When to Use Default and Named Segments 4-46

Considerations for Growing Tables 4-48
Effect of Table Growth on STAR Indexes 4-49

In This Chapter

This chapter describes the planning that must be done before you implement a Red Brick Decision Server database. It includes the following sections:

- Organizing Data into Databases
- Determining When to Create Additional Indexes
- Planning for TARGETjoin Processing
- Planning Disk Storage Organization
- Estimating Temporary Space Requirements
- Planning for Segmented Storage
- Considerations for Growing Tables

Organizing Data into Databases

In planning for a new Red Brick Decision Server installation, the database administrator must decide how user tables are to be organized into databases and, therefore, how many databases are to be created. A single database can contain many unrelated collections of user tables. In theory, every table in a large and complex installation can be created within a single database. In practice, however, it is usually desirable to separate distinct groups of user tables into separate databases. Separating tables into different databases eases administration; provides isolation for security, backup, and recovery; and makes the database appear less complex to end users.

In general, tables supporting a single business application should be located in a single database. This organization allows user-access permissions and macro definitions to be shared by the entire application. Conversely, tables supporting distinct business applications should be separated into distinct databases.

The first step in planning a new database is to identify and design the tables to be included in the database, specify their contents, and define the relationships among them. This process is discussed in Chapter 3, "Schema Design."

Determining When to Create Additional Indexes

In a Red Brick Decision Server database, a B-TREE index is automatically created on the primary key of a table during table creation. To improve query performance, create additional indexes. The improved query performance must be balanced against the additional storage space for each index and the additional time to build or update it during each load procedure. In general, if space is available and load performance is not an issue, index any column that is constrained in queries.

You can drop any index at any time. If you drop a primary key index, however, any INSERT, UPDATE, or DELETE operation on that table might result in referential integrity violations unless you build either another B-TREE index on the primary key or a STAR index.

If your database contains any outboard tables, create a B-TREE index or a STAR index on each foreign key of each table referencing an outboard table.

Creating additional indexes can have a large impact on query performance. For information about how Red Brick Decision Server chooses the indexes to use in a query and the algorithms used to join tables, refer to "Understanding Query Processing" on page 10-44.

Consider the following guidelines about creating additional indexes:

- If your schema design is a star schema in which you have a central referencing (fact) table with primary key-foreign key references to one or more referenced (dimension) tables, create one or more STAR indexes. If queries against a fact table constrain only the trailing foreign key column of an existing STAR index, or if queries constrain the trailing columns more tightly than the leading columns, create an additional STAR index naming the more tightly constrained columns as the leading columns in the index key to improve query performance. STAR indexes are usually more useful when they cover more columns. A single-column STAR index is not a very useful index.
- If your schema design is a star schema, if you have many referenced (dimension) tables, and if the STAR indexes you have created are not providing optimal performance for all of your queries, you can create TARGET indexes on the foreign key columns of the referencing (fact) table to enable TARGETjoin processing. For more information about TARGETjoin processing, refer to "Planning for TARGETjoin Processing" on page 4-12 and to Chapter 10, "Tuning a Warehouse for Performance."
- Queries against multiple fact tables (fact-to-fact table joins) place two specific requirements on STAR indexes:
  - The fact tables must each have at least one foreign key reference to at least one common dimension table.
  - All the shared foreign keys must appear in the same relative order in each STAR index.

  Create additional STAR indexes to satisfy these requirements in cases where queries against multiple fact tables are needed. If these conditions are not satisfied, the query proceeds using a different join algorithm and might not perform as well.
- Define B-TREE indexes on each UNIQUE column to enforce the uniqueness constraint.

- Create indexes on aggregate tables equivalent to those created on detail tables. For details, refer to the *Informix Vista User's Guide*.
- Queries against a table constrain columns other than the primary key:
  - Create a B-TREE index on each column that will be constrained to improve query performance unless that column contains a large number (20 percent or more) of duplicate values. (For example, do not create a B-TREE index on a column that has only a few possible values, such as YES, NO, or NA.)
  - Create a TARGET index on any columns that contain a large number of duplicate values.



Tip: *In all cases where a dimension table contains a foreign key (that is, it references an outboard table), create a B-TREE or STAR index on that foreign key.*

The examples that follow illustrate each of these cases.

STAR Indexes

Red Brick Decision Server uses STAR indexes to greatly improve performance on queries involving tables that have foreign keys that are the primary keys of another table. A STAR index uses STARjoin technology, a proprietary method of joining tables with a primary key / foreign key relationship in the schema design. When you have such schemas, known as star schemas, you must create one or more STAR indexes to take advantage of this technology.

If you have multiple STAR indexes, the order of the foreign key constraints in a query determines which index is used. Red Brick Decision Server uses the STAR index built on columns with the closest match to the query constraints.

Example

This example illustrates a case in which the leading foreign key columns of a STAR index are not constrained. An additional STAR index on the constrained columns will improve performance.

A fact table is defined as follows:

```
create table table1 (
pk int not null unique,
fk1 int not null,
fk2 char (3),
fk3 char(2),
fk4 char(6),
fk5 int,
col1 char(8),
col2 char(10),
constraint table1_pkc1 primary key (pk, fk1, fk2)
constraint tabled1_fkc3 foreign key (fk3)
    references tabled1 (pk),
constraint tabled2_fkc2 foreign key (fk2)
    references tabled2 (pk),
constraint tabled3_fkc1 foreign key (fk1)
    references tabled3 (pk),
constraint tabled4_fkc4 foreign key (fk4)
    references tabled4 (pk),
constraint tabled5_fkc5 foreign key (fk5)
    references tabled5 (pk)
) ;
```

The primary key B-TREE index is automatically built on columns **Pk**, **Fk1**, and **Fk2**. You can build a STAR index on the foreign key columns **Fk3**, **Fk2**, **Fk1**, **Fk4**, and **Fk5**, in that order, as follows:

```
create star index star1 on table1
(tabled1_fkc3, tabled2_fkc2, tabled3_fkc1, tabled4_fkc4,
   tabled5_fkc5) ;
```

This index provides good performance on queries that constrain on, for example, **Fk3**, **Fk2**, and **Fk1**. For queries that constrain on **Fk4** and **Fk5**, you can improve the performance by creating an additional STAR index on columns **Fk4** and **Fk5**, as follows:

```
dbspcreate star index star2 on table1 (tabled4_fkc4,
   tabled5_fkc5) ;
```

Example

This example illustrates a case in which a STAR index is used to perform an efficient join between two fact tables (fact-to-fact table joins). In this case, suppose you build a STAR index that includes all the foreign keys, in the order they are specified in the CREATE TABLE statement, for each table. The foreign key columns that make up the STAR indexes are not in the same order.

Two tables contain the following FOREIGN KEY clauses. The shared foreign key references are in bold typeface.

```
create table fact1 (
...
foreign key (fk1) references dimension1 (pk),
foreign key (fk2) references dimension2 (pk),
foreign key (fk3) references dimension3 (pk),
foreign key (fk4) references dimension4 (pk))

create table fact2 (
...
foreign key (fky1) references dimensionx (pk),
foreign key (fky2) references dimension3 (pk),
foreign key (fky3) references dimension2 (pk),
foreign key (fky4) references dimension1 (pk),
foreign key (fky5) references dimensiony (pk)) ;
```

The foreign key clauses that are common to both tables (the ones that reference tables **Dimension1**, **Dimension2**, and **Dimension3**) are not in the same order in both tables. Because the STAR indexes you already built are in the foreign key order specified in the CREATE TABLE statement, their keys are not in the same order, which is a requirement for joins between fact tables.

An analyst wants to write queries that join tables **Fact1** and **Fact2** and constrain on the tables **Dimension1** and **Dimension2**. A STAR index on the **Fact2** table defined as follows provides an index with the required key composition and order:

```
create star index star2 on fact2 (fky4, fky3, fky2) ;
```

The requirements listed in "STAR Indexes" on page 4-6 for STAR indexes used for fact-to-fact joins are met. All foreign keys shared by the fact tables are present in a STAR index for each table (the STAR index on all the foreign keys for **Fact1** and **Star2** on **Fact2**), and the shared keys are in the same relative order (**Dimension1**, **Dimension2**, **Dimension3**). If the analyst's queries constrain on only the shared dimension tables, the index **Star2** is sufficient.

If the analyst wants to constrain table **Fact2** on its nonshared foreign key **Fk5**, which references the table **Dimensiony**, that column must also be included in a STAR index.

```
create star index star3 on fact2 (fky4, fky3, fky5, fky2) ;
```

Relative but not identical order of the shared foreign keys is required. In defining key order, also consider frequency and selectiveness of constraints.

B-TREE Indexes

You can create a B-TREE index on any column or set of columns in a table. If you have a join query and no STAR index or TARGET index is available, but a B-TREE index is available, the B-TREE index is used, causing a nested loop join.



Tip: *If you create a multicolumn B-TREE index, all columns included in the index must be declared NOT NULL, or the index creation terminates with an error.*

Example

This example illustrates a case where a nonprimary key column in a table is constrained. An additional index improves performance.

A table named **Product** with an outboard table named **Personality** is defined as follows:

```
create table product (
prodkey integer not null,
product char (15),
distributor char (15),
beankey integer not null,
primary key (prodkey),
foreign key (beankey) references personality (beankey)) ;
```

An index is created automatically on **Prodkey**, the primary key column in the **Product** table (and also on the **Beankey** column of the **Personality** table, which is the primary key of that table). If queries constrain any of the other columns in the **Product** table, creating B-TREE indexes on those columns improves performance. For example, if you have queries that constrain on the **Distributor** column, you can create a B-TREE index as follows:

```
create index on product (distributor) ;
```

TARGET Indexes

TARGET indexes have two different applications:

- TARGET indexes on columns of referenced (dimension) tables that are constrained in queries.
- TARGET indexes on the foreign key columns of a referencing (fact) table to enable TARGETjoin processing.

If you use queries that contain multiple weakly selective constraints, creating TARGET indexes on the columns subject to these constraints can improve performance. Performance improvements are two-fold. The queries run faster and require less memory to process.

For information about using TARGET indexes to enable TARGETjoin query processing, refer to "Planning for TARGETjoin Processing" on page 4-12 and to Chapter 10, "Tuning a Warehouse for Performance."

Understanding Weakly Selective Constraints

The term *weakly selective* describes a constraint that retrieves many records from a table. Weak selectivity typically occurs when a column in a very large table has a small *domain* (set of possible values). For example, the domain of a **Gender** column in an **Employees** table consists of only two possible values for every row—`male` or `female`. Constraints on that column are weakly selective. They usually retrieve a very large list of rows.

Much larger domains might also give rise to weak selectivity. For example, an **Age** column in the same table would have a much larger domain than a **Gender** column, but constraints on age might still be weakly selective, especially if the data is not uniformly spread across the domain or if the constraints specify values that dominate the domain.

Choosing the Right Domain Size

When you create a TARGET index on a particular column and you know the data in the column you are indexing, you can specify the domain size—LARGE, MEDIUM, or SMALL—in the DOMAIN clause of the CREATE INDEX statement. Based on your choice, Red Brick Decision Server selects the appropriate storage method, or representation, for the TARGET index information.

If you do not specify the domain size when you create the TARGET index, Red Brick Decision Server dynamically selects the storage representation for each distinct key value (each different value in the column) of the TARGET index column. The storage representation could be different for different key values, depending on the number of occurrences of each distinct key value in the indexed column. This method works well when the data is not uniform and when you are not sure of the data in the column you are indexing.

For more information on the DOMAIN clause of the CREATE INDEX statement, refer to the *SQL Reference Guide*.

Knowing Your Data

If you know what your data is like, you can make a good choice of what type of TARGET index to create. If the data in a column is uniformly distributed and you have a good idea of the domain size, you can use the DOMAIN clause when creating your TARGET index. However, if the data is skewed, you do not have a good idea of the domain size, or both, you can create a TARGET index without specifying the DOMAIN clause. This TARGET index has a *hybrid* representation that dynamically chooses its domain size, based on the data.

An advantage of this hybrid type of TARGET index is that you do not need to choose a domain size, which is useful when you do not know what the data in the column will be like over time. A disadvantage is that you have less control over how the index grows with the data. It is more difficult to estimate how large the index might grow over time.

Example

Suppose you have a 10,000,000 row **Customer** table with the columns **Cust_key**, **Last_name**, **First_name**, **Street**, **City**, **State**, **Zip_code**, and **Region**, where **Region** has a value between 1 and 250, each number representing a particular sales representative's territory. You can create a TARGET index to improve query performance on queries constraining against the **Region** column as follows:

```
create target index customer_region_idx on customer (region)
   domain size medium;
```

No Indexes

If you join tables that do not have indexes covering the join path and the join is an equijoin, Red Brick Decision Server performs the join using a hybrid hash join algorithm. Hash joins are efficient for joining tables that are so different in size that the smaller table can fit into memory.

If the join is not an equijoin and no indexes are available, it can be performed as a cross join. The cross join finds the Cartesian product of the tables being joined and must be enabled using the SET CROSS JOIN ON statement. For more information on the SET CROSS JOIN statement, refer to the *SQL Reference Guide*.

Planning for TARGETjoin Processing

Red Brick Decision Server includes a family of join methods, one of which is the TARGETjoin bit-mapped join. TARGETjoin processing works on star schemas or any schema that has primary key / foreign key relationships. It is a complementary join method to STARjoin technology. It uses TARGET indexes on the foreign keys of a fact table (B-TREE indexes on multicolumn foreign keys) to join the table to the tables referenced by the foreign keys. This section explains how TARGETjoin processing works and provides information on how to use and administer a database to take advantage of this new join method. The following topics are included:

- STARjoin Versus TARGETjoin
- Administration Considerations for TARGETjoin Processing
- TARGET Index DOMAIN Clause

For more information about using TARGETjoin query processing to improve performance, refer to "TARGETjoin Query Processing" on page 10-61.

STARjoin Versus TARGETjoin

Red Brick Decision Server uses STAR indexes to provide fast joins between a fact table and the dimension tables it references with foreign key / primary key relationships. STARjoin processing also supports joins between two or more fact tables with related dimension tables. STAR indexes work well on queries that join fact and dimension tables, particularly when one or more of the leading keys of the STAR index are constrained in the query. When one or more of the leading keys of a STAR index are not constrained in a query, the performance of a STARjoin query is generally not as good as when they are constrained. As the constraining columns become separated by more and more columns from the leading key in the STAR index, the ability of the index to enhance the performance of the query diminishes. One alternative is to create additional STAR indexes with different key orders or with different keys. TARGETjoin processing offers another alternative.

When you submit a query to Red Brick Decision Server, the database server first generates a plan for execution based on the performance of each join method for the indexes available. The database server chooses between STARjoin, TARGETjoin, B-TREE 1-1 match, hybrid hash join, and cross join. The actual query path is chosen at query execution time. The database server chooses a TARGETjoin query plan when a good STARjoin query plan is not available and when the TARGETjoin operation will perform better than a table scan.

TARGETjoin processing is not a replacement for STARjoin processing. They are complementary technologies. All other things being equal, with a good STARindex (an index that contains the columns constrained in the query as leading keys in the STARindex), STARjoin processing is a faster join method than TARGETjoin processing. However, with a STARindex where the leading keys are not constrained in the query, TARGETjoin processing likely provides better performance.

Administration Considerations for TARGETjoin Processing

When you create TARGET indexes on fact table foreign keys (B-TREE indexes on multicolumn foreign keys) to enable TARGETjoin processing, you must consider the administrative costs associated with maintaining these new indexes. If your database is static—that is, it does not change through incremental INSERT, UPDATE, DELETE, or LOAD DATA operations— the cost of the indexes is the amount of extra disk space they require. The amount of disk space required is proportional to the number of rows in your fact table. A table with 100 million rows has much smaller indexes than a table with 4 billion rows. Use the **dbsize** tool to estimate index size.

If your database is dynamic, for instance if you perform incremental INSERT, UPDATE, DELETE, or LOAD DATA operations, you incur an additional cost in time to update these indexes while you change the database. Consider these costs and plan for them, putting the appropriate administrative procedures in place. Every implementation is unique, but this section discusses some general issues to consider for all databases.

Cost Versus Performance

Performance is the primary reason for using TARGETjoin processing. If your system is performing well in all situations already, you probably do not need to incur the extra cost of building the indexes on your foreign keys. STARjoin processing is extremely effective on many schemas. You simply might not need TARGETjoin processing. But if you have some queries that constrain on columns that are not among the leading keys of your STARindexes, consider adding the indexes to enable TARGETjoin processing.

Two types of costs are involved in creating the indexes to enable TARGETjoin processing:

- Increased disk space to store the indexes
- Increased load times to update the indexes

Load Operations

When the TMU loads data into a database that contains TARGET or B-TREE indexes on the foreign keys of the fact table, time is required to update these indexes during the load. Depending on the time available for loading the data (the load window), consider dropping these TARGET and B-TREE indexes before the load and then re-creating them after the load. The advantages of this approach are as follows:

- Faster load times because the TARGET and B-TREE indexes do not need to be built for the load operation to complete
- Increased database availability because the database is available for querying while the TARGET and B-TREE indexes are building

  Queries do not use TARGETjoin processing until the appropriate indexes become available. When the indexes become available, Red Brick Decision Server automatically considers them when compiling queries. The net gain is less downtime for the database for load operations although the foreign key indexes are not immediately availability.

Consider the following two cases regarding loading data:

- Load operations that roll off old data segments and roll on new ones
- Load operations that do not change the table structure

Load Operations That Roll Segments Off and On

If your database is segmented by time and you routinely roll off old data and roll on new data by using ALTER SEGMENT...DETACH OVERRIDE FULLINDEXCHECK and ALTER SEGMENT...ATTACH operations, dropping your foreign key indexes before you load and re-creating them after you load might be a good strategy. Because this type of load changes the structure of the table, thus invalidating any indexes that are not segmented in the same way as the data, the TARGET and B-TREE indexes would require a REORG operation. The extra time required to load and update the indexes and then to perform a REORG operation might be more costly than to drop the foreign key indexes, perform the load operation, and then re-create the indexes.

Incremental Loads That Do Not Change the Table Structure

An INSERT, UPDATE, DELETE, or LOAD operation that does not change the structure of any tables updates all of the indexes, including any TARGET indexes, during the operation. These operations do not require a REORG operation after the data is loaded. This approach is efficient in the following cases:

- INSERT, UPDATE, DELETE, or LOAD operations are relatively small.
- Databases are designed for growth by allocating segments for future loads when the tables are created.
- Database updates are infrequent.

The cost of these incremental changes grows as database size grows. If your database has millions of rows, these operations tend to be much less costly than if your database has billions of rows. In all cases, carefully evaluate the cost of these operations with all of your indexes in place.

Large Dimension Tables

If you have a schema with large dimension tables, consider the following issues about TARGETjoin processing. A typical example of a large dimension table is a customer table, but any dimension table with many rows (for example, more than 10,000) can be considered a large dimension table.

Queries that use TARGETjoin processing to join large dimension tables to a fact table present two potential difficulties:

- The TARGET and B-TREE indexes on the foreign keys might use a large amount of disk space.
- Queries that have weakly selective constraints on the large dimension table might not perform well.

The first concern is purely a resource problem. If you can afford the space and the time needed to create the index, this is not a problem. The index size and the time to create it vary depending on your schema. Use the **dbsize** tool to estimate the index size.

The second concern occurs when you join the large dimension table to the fact table with a query that has a large number of qualifying rows (weakly selective) on the dimension table. For example, suppose you have a customer table with 1,000,000 rows. Suppose also that you want to know something about all of your customers who are male, and 75 percent of your customers are male. That means 750,000 rows of the dimension table qualify for this join. In this case, TARGETjoin processing does not perform well. (STARjoin processing performs well on these weakly selective constraints on large dimensions when there is a STARindex in which one of the *trailing* keys is the foreign key corresponding to the large dimension table.)

However, if your query constraint on the **Customer** table produces a small number of qualifying rows, that query will perform well using TARGETjoin processing. For example, suppose a query has the following constraint:

```
where customer.customer_last_name like 'X%'
```

Only one customer has a last name that begins with X, so this query performs well using TARGETjoin processing.

Multicolumn Foreign Keys

When you have a schema that contains one or more multicolumn foreign keys on a fact table, and you want to use TARGETjoin processing to join the fact table to the dimension table referenced by the multicolumn key, create a single B-TREE index on the concatenation of all of the columns of the multi-column foreign key.

For example, the **Sales** table in the **Aroma** database contains a two-column foreign key, **classkey** and **prodkey**, that references the **Product** table. To allow TARGETjoin processing between the **Sales** and **Product** tables, you must create a B-TREE index on the **Sales** table with a CREATE INDEX statement such as the following one:

```
create index sales_classkey_prodkey_btree_idx
      on sales (classkey, prodkey);
```

For information about the CREATE INDEX statement, refer to the *SQL Reference Guide*.



Tip: *The performance of a TARGETjoin query with multicolumn foreign keys is generally not as good as with single-column foreign keys. If possible, design schemas that have single-column foreign keys for the best performance.*

Parallel TARGETjoin Queries

Like other queries in Red Brick Decision Server, TARGETjoin queries use parallel processing when it is appropriate. The same rules apply to TARGETjoin parallel queries that apply to STARjoin parallel queries. The parallel tuning and SET parameters that apply to parallel STARjoin queries also apply to parallel TARGETjoin queries and are as follows:

- ROWS_PER_FETCH_TASK
- FORCE_FETCH_TASKS

For information about setting these parameters and about parallel queries, refer to Chapter 10, "Tuning a Warehouse for Performance."

TARGET Index DOMAIN Clause

When you create a TARGET index, the DOMAIN clause is optional. If you omit this clause, Red Brick Decision Server dynamically chooses a hybrid representation for each distinct key value (each different value in the column) based on the data. The storage representation could be different for different key values, depending on the number of occurrences of each distinct key value in the indexed column. This method works well when the data is not uniform and when you are not sure of the data in the column you are indexing.

When deciding whether to include the DOMAIN clause in a TARGET index, consider what you know about your data. If you have a good idea of the level of uniformity of your data, you can probably make a good choice as to the DOMAIN size. If you do not know your data that well, or if you are unsure, do not specify the DOMAIN clause. This automatic hybrid representation works well in most cases.

Use the *dbsize* utility to estimate the size of TARGET indexes. The size will change depending on the DOMAIN clause specification of SMALL, MEDIUM, or LARGE. To estimate the range of sizes of a TARGETindex where the DOMAIN clause is not specified (the hybrid representation), use **dbsize** to calculate the size of indexes of all three DOMAIN values. The upper and lower bounds from these results define the range of sizes.

The following table shows the different domain sizes, the representation that each type uses to store data, and recommendations on when to use each type.

DOMAIN Size	Representation	When to Use
Not specified	Hybrid; changes with data	Use when you do not know your data, when data is skewed, or when you are not sure which DOMAIN size to choose. This is the default and is generally a good choice.
SMALL	Bitmap	Use when you have fewer than 100 distinct values or when the expected number of rows in the fact table for each distinct foreign key value in the fact table is high (greater than 100,000). This domain size tends to offer the best performance but can also use the most disk space when the number of distinct values is high.
MEDIUM	Compressed row list	Use when the number of distinct values is between 100 and 1000 or when the expected number of rows in the fact table for each distinct foreign key value in the fact table is medium (between 1000 and 100,000).
LARGE	Uncompressed row list	Use when the number of distinct values is greater than 1000 or when the expected number of rows in the fact table for each distinct foreign key value in the fact table is low (less than 1000).

When you consider the number of unique values, the important number is the number of unique values in the fact table. This number might be smaller than the number of unique values in the dimension table. For example, a **Day** table might contain ten years worth of days, or 3650 unique values, but the database (and therefore the fact table) might contain only one year of data, or 365 unique days. In this case, the best choice for a TARGET index on the foreign key column that references the **Day** table is a DOMAIN MEDIUM TARGETindex.

For the complete syntax for creating TARGET indexes, refer to the CREATE INDEX statement in the *SQL Reference Guide*.

Planning Disk Storage Organization

Before creating a new database, carefully estimate the disk space requirements for the database and its contents so that you can decide whether to organize the database contents entirely into default segments or to use named segments for some or all tables or indexes. To estimate disk space requirements, you need to estimate the sum of the space required for each permanent and temporary user table, each automatic and optional index, the system tables, and the expected growth patterns of your tables.

More specifically, you need to identify the following:

- The specific permanent user tables to be included in the database.
- The maximum number of temporary user tables that might exist within the database at any given time.
- For every permanent and temporary table, the initial number of rows to be loaded or inserted into the table.
- For every table, the data types and sizes of every column. For VARCHAR columns, you need to know the maximum size of the column and the typical size of the data that will go into the column.
- For every table, the columns to be indexed and the types of indexes to be used. Consider automatic indexes (primary key B-TREE indexes) and optional indexes (additional B-TREE indexes, STAR indexes, and TARGET indexes).
- Anticipated growth patterns for all tables, including growth rate and maximum expected number of rows, within your planning horizon.

Using the information from your schema design, perform the following steps to determine the appropriate disk space organization. Each step is described in the sections that follow.

1. Use the **dbsize** utility to estimate the storage space for each user table to be included in the database.
2. Use the **dbsize** utility to estimate the storage space for each automatic and optional index to be included in the database.

3. Use the **dbsize** utility to estimate the size of the system tables for the database.
4. Determine how you want to use segmented storage, planning a strategy for distributing data across multiple disks if necessary. If any of your user tables are expected to grow (in terms of number of rows stored), review the discussion about growing tables on page 4-48.

Estimating the Size of User Tables

To calculate the size of a user table, you must know the following:

- The number of columns
- Data type of each column
- The fill factor for variable-width (VARCHAR) columns
- The expected number of rows in each table

From this information, you can use the **dbsize** utility, which is included with Red Brick Decision Server, to calculate the length in bytes of each row and the number of bytes required to store all the rows. On UNIX, this utility is located in the **/redbrick_dir/util/service** directory. On Windows, it is located in the **\redbrick_dir\util\service** directory. For information about this utility, refer to the **README** file, also in the **service** directory.

For information on setting VARCHAR column fill factors for maximum performance, see "Setting the VARCHAR Column Fill Factor" on page 10-29.

For temporary tables for which multiple instances might be present in the database at the same time, multiply the estimated size of a single instance of the temporary table by the expected maximum number of instances of that table that will exist within the database at any given time. Use this product as the estimated size for the temporary table. For more information, refer to "Estimating Temporary Space Requirements" on page 4-35.

Example

Assume a table is created as follows:

```
create table fact_table(
   prodkey integer not null,
   mktkey integer not null,
   description character(65),
   dollars decimal (12,2),
   primary key (mktkey, prodkey),
   foreign key (mktkey) references market (mktkey),
   foreign key (prodkey) references product (prodkey));
```

Assume the table will contain approximately 53,000,000 rows:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the data types of the columns, the precision of the decimal data type, and the total number of estimated rows for the table.
4. For the preceding CREATE TABLE statement, **dbsize** estimates the size of the table to be 4,156,872 kilobytes.

Estimating the Size of Indexes

To estimate the amount of space required by indexes in the database, you must know the following information:

- How fill factors determine the amount of space used in STAR and B-TREE index nodes
- How many indexes exist for each table in the database
- How to estimate the size for each type of index: STAR, B-TREE, and TARGET

Index Fill Factors

Each index has a fill factor associated with it. The index fill factor is the percentage to which an index node is filled on *initial creation* of that node. Initial creation of a node occurs in the following three instances:

- During the initial load operation when the primary key indexes are built
- Whenever indexes are created with the CREATE INDEX statement
- During incremental load operations when new nodes are created because the previous node reached the fill factor

After an index node is created and filled to the level specified by the fill factor, subsequent load operations that insert entries into existing nodes fill those nodes 100 percent full before a full node splits to form two new 50-percent-full nodes. Therefore, some nodes might be fuller than specified by the fill factor. Each new node must then fill to 100 percent before it splits again.



Important: *If you do not use optimize mode with the TMU, any specified fill factor is ignored.*

The purpose of the fill factor is to reserve extra space in each node. The extra space is used for incremental load operations that insert entries into existing nodes. If there is sufficient space in the existing nodes for the load operation to complete, the nodes do not have to split, which is a time-consuming operation. Eventually, the nodes might become full and split, but the fill factor gives you some control over the occurrence of these splits, thereby improving the performance of incremental load operations.

Leaving space for future entries in index nodes has the following costs:

- The additional space is allocated but not immediately used in each index node.
- More time is required to traverse deep, sparsely filled nodes than to traverse shallower, more densely filled nodes.

Fill Factors in Index Size Estimates

When estimating the size of an index, you might want to run the calculation several times to get an idea of how big your index will be initially and how big it might grow.

Because index size depends on the data to be inserted and the order of insertion, calculating the exact minimum and maximum size of an index is impossible. However, by estimating the size of an index several times using a different fill factor each time, you have a better idea of the space required as tables and indexes change over time.

As a guideline, assume that a typical index (one that undergoes an average amount of insertion and deletion after it is initially built) will generally be between 66 percent and 75 percent full. Use the following descriptions to select fill factors for your STAR and B-TREE index size calculations, using multiple calculations to provide a realistic size range for each index that will change:

- An arbitrary fill factor tells you how much space the index would take if it were a new index built from scratch in optimize mode with that fill factor (CREATE INDEX or LOAD / REORG in optimize mode).
- A fill factor of 100 percent tells you the absolute minimum amount of space the index would take.
- A fill factor of 75 percent gives you an approximate lower bound on the size of the index after it has undergone an average amount of change (after the operation that initially built it).
- A fill factor of 66 percent gives you an approximate upper bound on the size of the index after it has undergone an average amount of change (after the operation that initially built it).
- A fill factor of 50 percent tells you the practical maximum for the amount of space the index would take (unless you build your index with a fill factor of less than 50 percent, in which case that fill factor should be used to determine the practical maximum).

Use the following guidelines to choose an index fill factor:

- If you do not plan to perform incremental load operations but intend to load your database completely each time you update it, specify a high fill factor. You do not need to save room in the index nodes to insert more entries.
- If you are initially loading only a fraction of the data you anticipate loading in the time frame for which you are planning, specify a fill factor that corresponds to the percentage of data in the initial load. You want to save space in each index node for entries corresponding to the remaining data.

For example, if you are loading 95 percent of the data in the initial load, with a relatively small amount of data to be added later, specify a 95-percent fill factor. Conversely, if you are loading only 5 percent of the data initially before you load the remaining data for the production database, specify a 5 percent fill factor.



Important: *Fill factors are specified in the* ***rbw.config*** *file for the automatically created indexes and with a CREATE INDEX statement for user-created indexes. The fill factors for a specific index can be modified with an ALTER INDEX statement.*

For information on setting or changing an index fill factor, or finding the fill factor for a specific index, refer to "Setting the Index Fill Factor" on page 10-38.

STAR Indexes

The size of a STAR index depends on the expected number of rows in the tables referenced by the index. If you know this number, use it to determine the MAXROWS PER SEGMENT and MAXSEGMENTS values when you create the table. You must specify MAXROWS PER SEGMENT when you create referenced (dimension) tables that will participate in a STAR index. Otherwise, the CREATE STAR INDEX statement fails. By accurately forecasting and explicitly specifying the maximum number of rows when the table is created, you simplify size calculations and maintenance tasks, such as rebuilding STAR indexes when the dimension tables grow.

To estimate the size of a STAR index using the **dbsize** utility, you need to know the following:

- For each foreign key column that participates in the STAR index, the estimated maximum number of rows and number of segments in the referenced (dimension) table (MAXROWS PER SEGMENT multiplied by MAXSEGMENTS)
- The estimated number of rows to be included in the referencing (fact) table
- The fill factor for the index

B-TREE Indexes

To estimate the size of a B-TREE index using the **dbsize** utility, you need to know the following:

- The number of columns to be indexed
- The data type of each column to be indexed
- The fill factor for the index
- The estimated number of rows to be included in the indexed table

TARGET Indexes

To estimate the size of a TARGET index using the **dbsize** utility, you need to know the following information:

- The estimated domain size (the number of possible unique values) for the indexed column
- The data type of the column in the TARGET index
- The estimated number of rows (MAXROWS PER SEGMENT) in each segment of the indexed table
- The estimated percentage of NULL rows in the indexed table
- The domain for the column: SMALL, MEDIUM, or LARGE
- The number of segments in the table

Example: Calculating Table, Index, and System Table Sizes

This example illustrates how to size a database. Assume the following tables will be included in a new database.

Table **Fact1** contains approximately 53,000,000 rows and is defined as follows:

```
create table fact1 (
   tran integer not null,
   seq integer not null,
   prodkey integer not null,
   mktkey integer not null,
   description character(55),
   dollars decimal(12,2),
   primary key (tran, seq),
   foreign key (mktkey) references market (mktkey),
   foreign key (prodkey) references product (prodkey));
```

A STAR index will be built on the **Prodkey** and **Mktkey** columns.

Table **Market** contains approximately 5000 rows and is defined as follows:

```
create table market (
   mktkey integer not null,
   mktname character(20),
   primary key (mktkey));
```

A TARGET index will be built on the **Mktname** column.

Table **Product** contains approximately 520,000 rows and is defined as follows:

```
create table product (
   prodkey integer not null,
   category integer,
   primary key (prodkey));
```

A B-TREE index will be built on the **Category** column.

Fact1 Table and Its Indexes

The table **Fact1** has both a STAR index and a primary key B-TREE index.

Table Size: Fact1

Use **dbsize** to calculate table size for the table **Fact1**:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the data types of the columns, the precision of the decimal data type, and the total number of estimated rows for the table, which is `53,000,000` in this example.
4. For the CREATE TABLE statement for **Fact1**, **dbsize** estimates the size of the table to be 4,038,104 kilobytes.

STAR Index: Fact1

Use **dbsize** to calculate the STAR index size. Table **Fact1** has foreign key references to tables **Market** and **Product**, and you have decided to create a STAR index on the **Prodkey** and **Mktkey** columns:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of STAR indexes.
4. Enter the maximum number of rows for the **Market** and **Product** tables.
5. Enter the estimated number of rows for the fact table, which is `53,000,000` in this example.
6. Enter the fill factor. Assume the data is ordered for the initial load but will be updated through incremental loads, so use a fill factor of 66 percent. For a fill factor of 66 percent, enter `.66` in **dbsize**.
7. For the CREATE TABLE statement for **Fact1**, **dbsize** estimates the size of the STAR index to be 946,448 kilobytes.

Primary B-TREE Index: Fact1

The primary key B-TREE index is created automatically when you create the table **Fact1**. Use **dbsize** to calculate the B-TREE index size:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of B-TREE indexes.
4. Enter the number of columns to be indexed, which is 2 (**Tran** and **Seq**) in this example.
5. Enter the data type for each column.
6. Enter the estimated number of rows for the table, which is `53,000,000` in this example.
7. Enter the fill factor. Assume the data is ordered for the initial load but will be updated through incremental loads, so use a fill factor of 66 percent. For a fill factor of 66 percent, enter .66 in **dbsize**.
8. For the CREATE TABLE statement for **Fact1**, **dbsize** estimates the size of the primary key B-TREE index to be 1,265,704 kilobytes.

Market Table and Its Indexes

The table **Market** has both a primary key B-TREE index and a TARGET index.

Table Size: Market

Use **dbsize** to calculate table size for the table **Market**:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the data types of the columns, and the total number of estimated rows for the table, which is `5000` in this example.
4. For the CREATE TABLE statement for **Market**, **dbsize** estimates the size of the table to be 136 kilobytes.

The **Market** table has a primary key B-TREE index and an optional TARGET index.

Index Size: Primary Key B-TREE Index

Use **dbsize** to calculate the size of B-TREE indexes:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of B-TREE indexes.
4. Enter the number of columns to be indexed, which is `1` (**Mktkey**) in this example.
5. Enter the data type for the column, which is `integer` for this example.
6. Enter the estimated number of rows for the indexed table, which is `5000` in this example.
7. Enter the fill factor. Assume the data is ordered and updated infrequently, so use a fill factor of 100 percent. For a fill factor of 100 percent, you must enter `1.00` in **dbsize**.
8. For the CREATE TABLE statement for **Market**, **dbsize** estimates the size of the primary key B-TREE index to be 80 kilobytes.

Index Size: TARGET Index on Mktname Column

Use **dbsize** to calculate the size of TARGET indexes:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of TARGET indexes.
4. Enter the estimated domain size for the **Mktname** column. In this example, assume that there are 20 markets and therefore a domain size of `20`.
5. Enter the data type for the column, which is `character` for this example.
6. Enter the number of characters, which is `20` in this example.
7. Enter the estimated number of rows for the table, which is `5000` in this example.
8. Enter the estimated percentage of null rows for the **Market** table. For this example, assume no null values, and enter `0.0`.

9. Enter the domain size, which is `SMALL` in this example. For a description of domain size for TARGET indexes, refer to "TARGET Indexes" on page 4-10.
10. Enter the number of segments in the table, which is `1` for this example.
11. For the CREATE TABLE statement for **Market**, **dbsize** estimates the size of the TARGET index to be 200 kilobytes.

Product Table and Its Indexes

The **Product** table has a primary key B-TREE index and an optional B-TREE index.

Table Size: Product

Use **dbsize** to calculate table size for the table **Product**:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of user tables.
3. Respond to the prompts for the number of columns, the data types of the columns, and the total number of estimated rows for the table, which is `520,000` in this example.
4. For the CREATE TABLE statement for **Product**, **dbsize** estimates the size of the table to be 4584 kilobytes.

The **Product** table has a primary key B-TREE index and an optional B-TREE index.

Index Size: Primary Key B-TREE Index

Use **dbsize** to calculate the size of B-TREE indexes:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of B-TREE indexes.
4. Enter the number of columns to be indexed, which is `1` (**Prodkey**) in this example.

5. Enter the data type for the column, which is `integer` for this example.
6. Enter the estimated number of rows for the fact table, which is `520,000` in this example.
7. Enter the fill factor. Assume the data is updated frequently, so use a fill factor of 66 percent. For a fill factor of 66 percent, enter `.66` in **dbsize**.
8. For the CREATE TABLE statement for **Product**, **dbsize** estimates the size of the primary key B-TREE index to be 9312 kilobytes.

Index Size: B-TREE Index on Category Column

Use **dbsize** to perform the following actions:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of indexes.
3. Choose the option to estimate the size of B-TREE indexes.
4. Enter the number of columns to be indexed, which is `1` (**Category**) in this example.
5. Enter the data type for the column, which is `integer` for this example.
6. Enter the estimated number of rows for the fact table, which is `520,000` in this example.
7. Enter the fill factor. Assume the data is updated frequently, so use a fill factor of 66 percent. For a fill factor of 66 percent, enter `.66` in **dbsize**.
8. For the CREATE TABLE statement for **Product**, **dbsize** estimates the size of the B-TREE index on the **Category** column to be 9312 kilobytes.

Estimating the Size of System Tables

A complete set of system tables is created and maintained for each database. These tables contain information about the database and its users. The system tables are stored within the files named **RB_DEFAULT_IDX**, **RB_DEFAULT_INDEXES**, **RB_DEFAULT_LOADINFO**, **RB_DEFAULT_LOCKS**, **RB_DEFAULT_SEGMENTS**, and **RB_DEFAULT_TABLES**, which are located in the database directory.

The size of the system tables for a given database depends on the number of tables and columns created in the database, the number of views and macros defined, the number of users granted access to the database, and the load activity. The **RB_DEFAULT_IDX** and **RB_DEFAULT_LOADINFO** files grow as necessary to hold the system tables. For most databases, the total space required for the system tables is less than 1 megabyte.

Use **dbsize** to estimate the size of the system tables. This utility prompts you for the following information:

- The total number of columns
- The total number of indexes
- The total number of segments
- The total number of views
- The total number of tables
- The total number of primary keys and the number of columns each key contains
- The total number of foreign keys and the number of columns each key contains

Size: System Tables

For the database used in the previous examples, use **dbsize** to calculate the size of the system tables:

1. Run the **dbsize** utility.
2. Choose the option to estimate the size of system tables.
3. Enter the total number of columns, which is `10` in this example.
4. Enter the total number of tables, which is `3` in this example.
5. Enter the total number of segments. For this example, assume that each table uses 1 segment and that each index uses 1 segment. Therefore, there are 9 segments for this example.
6. Enter the total number of indexes, which is `6` in this example.
7. Enter the total number of views, which is `0` in this example.
8. Enter the number of primary keys plus .25 for each extra column in a multicolumn key. In this example, there are 3 primary keys, one of which is 2 columns. Therefore, enter `3.25`.

9. Enter the number of foreign keys, which is 2 in this example.
10. For this example, **dbsize** estimates the size of the system tables to be 392 kilobytes.

Total Space for User Tables, Indexes, and System Tables

Find the sum of the size estimates obtained in the previous examples from **dbsize** to calculate the total space required for all the user tables, indexes, and system tables.

Database Object	Space Required (in kilobytes)
Fact1 table	4,038,104
STAR index	946,448
Primary B-TREE index	1,265,704
Market table	136
Primary B-TREE index	80
TARGET index	200
Product table	4584
Primary B-TREE index	9312
B-TREE index on **Category**	9312
System tables	392
Total space	**6,274,272**

Estimating Temporary Space Requirements

In addition to the space needed to store the database tables and indexes, you also need to plan for temporary storage of intermediate results during optimized index-building and query operations. Temporary space for index building is controlled by the INDEX_TEMPSPACE configuration parameters. For queries, it is controlled by the QUERY_MEMORY_LIMIT and QUERY_TEMPSPACE configuration parameters. These configuration parameters specify one or more directories in which to store temporary files, how memory is used, and a maximum amount of disk space to be used as temporary space by a single operation.

This section describes temporary space requirements in terms of:

- How temporary space is used for optimized index-building operations, including those that build multiple indexes in parallel, both online and offline, and how to calculate the requirements for these operations
- How temporary space is used for queries and how to calculate the requirements

In addition to the information that follows about how various database operations use temporary space, you must also consider system resources and workload requirements. Some general considerations are as follows:

- Multiple databases

  Each database should have its own temporary space.
- Separate temporary-space areas for query and index-building operations

  If you use the same space for both types of operations, the system will dynamically and impartially allocate the space as needed. However, you might want more control over the resource allocation.
- Accommodating multiple users

  You can divide available space among multiple users, or you can plan to distribute the workload over time.

UNIX

- Maximum data segment size

  This operating-system value, usually set as a kernel configuration parameter, determines how much memory a process can use. This memory space is used by Red Brick Decision Server as buffer cache for the TMU (specified by the TMU_BUFFERS parameters), staging arrays for index building (specified by the parameter INDEX_TEMPSPACE_THRESHOLD), and the remainder as general working space. ♦

WIN NT/2000

- Physical memory

  Consider the amount of memory on your computer. ♦

- Swap space

  Even though a system is configured with sufficient swap space, out-of-memory errors can occur if insufficient swap space is available for the process stack and the data when a swap occurs.

For information about how temporary disk space is allocated and the temporary space parameters, refer to "Setting Temporary Space Parameters" on page 10-7.

How Optimized Index-Building Operations Use Temporary Space

Database administration operations that build an index for the first time use an optimized mode for building indexes from unordered data. Other operations that affect existing indexes (TMU LOAD DATA operations in APPEND, REPLACE, and INSERT mode and REORG operations) use optimized mode only if it is specified. The optimized method achieves faster performance and usually results in more compact indexes than those built in nonoptimized mode, but it achieves these improvements through more intensive use of memory and temporary disk storage (scratch space). As a result, these operations require additional planning to ensure adequate memory and disk space.



Tip: *Load operations that build a completely new index or completely replace an existing index from ordered data do not use optimized mode and hence have no special temporary space requirements. Information in this section does not apply to such operations.*

When optimized operations for data loads, table reorganization, and index building need to process a large number of rows, they do so by dividing the process into two phases. During the first phase—the sort phase—groups of index entries are built in the temporary space. During the second phase—the merge phase—these groups are merged into the index. Very large online load operations or the creation or reorganization of an index on a very large table often requires multiple cycles of sorting and merging.

During the sorting phase, the groups of index entries are first held in memory before being written to temporary disk space. The INDEX_TEMPSPACE THRESHOLD parameter determines the total amount of memory available to a server or the TMU for holding index entries. As the in-memory holding area fills up, groups are written to a file in the locations specified by the INDEX_TEMPSPACE DIRECTORY parameters or to system temporary space if this parameter is not set. The shift from the sort phase to the merge phase occurs when no more input data remains or when the limit on the amount of index-building temporary disk space (MAXSPILLSIZE) is reached. If multiple indexes are being built in a single operation, this space is divided among them.

Planning for these operations consists of making sure that adequate memory and disk space are available for them and setting the INDEX_TEMPSPACE parameters to reflect the available memory and disk space. Some guidelines for setting these parameters are provided in the following sections.

Estimating Temporary Space Values for Index-Building Operations

Index-building temporary space is used for load operations, table reorganization, and index creation that use the optimized index-building procedure. For operations that involve multiple indexes, the space is allocated evenly among all the indexes.

DIRECTORY Location Values

Temporary space of index-building operations can use directories spread over multiple file systems. The goal in specifying temporary space directories for INDEX_TEMPSPACE is two-fold: to overcome file system limitations and to avoid I/O contention. For each operation, the directories are used in a random sequence, with space being allocated only as it is needed.

If a warehouse contains multiple databases, each database should have a separate set of temporary space directories, which must be specified with SET commands rather than a single set of **rbw.config** file entries.

For more information about how space is allocated among the various directories designated as temporary space directories, refer to "Setting Temporary Space Parameters" on page 10-7.

THRESHOLD Value

The threshold specifies how much memory is used before the data spills to temporary space on disk. The goal in selecting a threshold for INDEX_TEMPSPACE is to select a large value relative to the system on which Red Brick Decision Server is running but not so large a value that errors occur. Selecting too large a value—a value that exceeds available memory—might cause the operation to fail with an out-of-memory error. Selecting too small a value might cause poor performance because of time spent writing to disk. You want to estimate a reasonable threshold for both online and offline load operations and other index-building operations.

UNIX

To estimate the threshold

1. Determine the maximum program data space allowed on your computer. This value is usually set as part of the UNIX kernel configuration for your computer (sometimes called *maxdsize*). A typical value is 64 megabytes. If you do not know the value configured for your computer, consult your system administrator or system vendor.
2. Select a value for INDEX_TEMPSPACE_THRESHOLD that is one quarter of the maximum data space size for your system. For example, if the maximum data space size on your computer is 64 megabytes, choose a value of 16 megabytes. (The value you enter is automatically rounded up to the nearest 8-kilobyte block.) ♦

WIN NT/2000

To estimate the threshold

1. Determine the amount of physical memory on your computer.
2. Select a value for INDEX_TEMPSPACE_THRESHOLD that is one quarter of the physical memory. For example, if the physical memory on your computer is 64 megabytes, choose a value of 16 megabytes. (The value you enter is automatically rounded up to the nearest 8-kilobyte block.) ♦

MAXSPILLSIZE Value

The maximum spill size specifies how much temporary disk space an operation can use. The goal in selecting a maximum spill size for INDEX_TEMPSPACE is to pick the largest value that allows the operation to complete without running out of temporary space. Selecting too large a value can cause a load operation, table reorganization, or index creation to fail with an out-of-space condition. Selecting too small a value for an online load increases the number of sorting-and-merging cycles performed. As a result, it increases the time required for the operation. Selecting too small a value for an offline load operation causes the operation to fail.

Because online operations can perform multiple sort and merge cycles, but offline operations must be able to complete in a single cycle, the maximum spill size requirements for online and offline operations are different.

Online Index-Building Operations

To determine a reasonable value for maximum spill size for online optimized index-building operations:

1. Determine the number of indexes, excluding TARGET indexes of DOMAIN SMALL or DOMAIN MEDIUM, that are to be processed in the operation. The maximum spill size will be split evenly among all these indexes.

   When determining the number of indexes involved in a load or REORG operation on a table, include all indexes (excluding TARGET indexes of DOMAIN SMALL or DOMAIN MEDIUM) on the table to be loaded.

   When determining the number of indexes involved in a REORG operation to rebuild one or more specific indexes by name, include the number of indexes (excluding TARGET indexes of DOMAIN SMALL or DOMAIN MEDIUM) named in the REORG statement.

   When determining the number of indexes involved in a CREATE INDEX operation, include the number of indexes (excluding TARGET indexes of DOMAIN SMALL or DOMAIN MEDIUM) being created in one statement (because each index is handled by a separate process).
2. For each index from step 1, use the **dbsize** utility to estimate the size of each index.

3. Add the values calculated with **dbsize** for each index (excluding TARGET indexes of DOMAIN SMALL or DOMAIN MEDIUM). This sum is the *total_space_required* for the index building operation.
4. Decide how much temporary disk space—MAXSPILLSIZE—to allocate for index building, based on system resources and the following relationships:

   ```
   total_space_required <= MAXSPILLSIZE x number_of_cycles

   MAXSPILLSIZE <= available_temp_space
   ```

number_of_cycles	The number of sort-and-merge cycles
available_temp_space	The space available in the directories designated as index-building temporary space

   If you have enough temporary space, a higher value of INDEX_TEMPSPACE_MAXSPILLSIZE results in a lower number of sort-and-merge cycles required to build the indexes, which ultimately results in a faster index-building time. The value you enter for INDEX_TEMPSPACE_MAXSPILLSIZE is automatically rounded up to the nearest 8-kilobyte block.

Offline Index-Building Operations

An offline load operation splits the two-phase processing (sorting and merging) of optimized loading into two separate TMU operations. An offline load operation on a large, multisegment table provides better database availability because users can still access the table for queries during the first phase of an offline load operation. During the actual offline load step, the TMU reads and formats the input records and loads them into the offline segment. It also builds the groups of index data in the index-building temporary space. During the second phase, a TMU SYNCH operation synchronizes the offline segment with the associated table.

A successful offline load operation requires more careful planning than online load operations for two reasons:

- For an offline load operation, enough temporary space must be available to complete the load operation in one pass. Otherwise, the operation fails. Unlike an online load, an offline load cannot cycle back and forth between building the index groups and merging them into the index. Thus, the amount of temporary space available and the setting of the INDEX_TEMPSPACE_MAXSPILLSIZE parameter limit how much data can be loaded in a single load step.
- The temporary space allocated by the offline load is not released until the SYNCH operation completes. For example, if the load operation is run during the day, and the SYNCH operation is delayed until evening to increase the daytime availability of the target table, the temporary space used by the load is unavailable for other operations during that time. If other load, table reorganization, or index-creation operations are performed during this interval, they might not find enough temporary space. As a result, you might want to create one or more temporary-space directories for an offline load operation and use the TMU SET command to set the INDEX_TEMPSPACE directories to those locations for that operation.

To calculate the disk space requirements for an offline load operation

1. For each index (including TARGET indexes), use the **dbsize** utility to estimate the size of each index.
2. Add the values calculated with **dbsize** for each index. This sum is *total_space_required* for the index-building operation.
3. Set MAXSPILLSIZE according to the following relationships:

   ```
   total_space_required <= MAXSPILLSIZE <=
   available_temp_space
   ```

   available_temp_space — The space available in the directories allocated for index-building temporary space

   The value you enter is automatically rounded up to the nearest 8-kilobyte block.

If you cannot set the INDEX_TEMPSPACE_MAXSPILLSIZE parameter to satisfy this relationship, you must either reduce the number of rows to be loaded in the offline load operation or increase the amount of temporary space available for this operation.

Temporary Space Requirements for TARGET Indexes

When you create TARGET indexes on large tables (for example, when you create TARGET indexes on foreign key columns of a fact table to enable TARGETjoin processing), the following temporary space requirements apply:

- For DOMAIN SMALL and DOMAIN MEDIUM, no temporary space is used.
- For DOMAIN LARGE and when no domain value is specified (the hybrid representation), the index build operation can use a *maximum* amount of temporary space based on the following formula:

  ```
  Temporary Space = (keysize + 11) x rows
  ```

keysize	The width (in bytes) of the column being indexed
rows	The number of rows in the table

  You do not necessarily need to allocate this much space for the index build operation, however. If the temporary space is exhausted, it triggers a sort-and-merge cycle from the temporary space into the actual index. The more sort-and-merge cycles that occur, the more time the index build operation takes.

  For example, if you were creating a hybrid TARGET index on an integer column of a billion row table, the *maximum* amount of temporary space Red Brick Decision Server will use to build the index is as follows:

  ```
  (4 + 11) x 1,000,000,000 = 15,000,000,000 bytes, or
  approximately 15 gigabytes
  ```

  In this case, allocating 15 gigabytes of temporary space ensures that the index will require only one sort-and-merge cycle, thus providing the fastest index build time. If you allocate 2 gigabytes of temporary space, this amount will trigger seven sort-and-merge cycles, adding to the total time of building the index.

How Query Operations Use Temporary Space

The amount of memory allocated to a query is set with the parameter QUERY_MEMORY_LIMIT. The location and amount of temporary space is allocated for query operations with the QUERY_TEMPSPACE_DIRECTORY and QUERY_TEMPSPACE_MAXSPILLSIZE parameters. This space is used to store the staging arrays for intermediate query results, subquery results, and final answer sets. It is used only by query operations and never by the TMU.

The query temporary space, like that for index-building operations, can be spread over directories residing in different file systems. The entire memory limit and maximum spill size values specified are allocated on a per-query basis, where a query is defined to include each of its subqueries, if any. For more information about how space is allocated among the various directories designated as temporary space directories, refer to "Setting Temporary Space Parameters" on page 10-7.

A query that exceeds the memory limit and spills to temporary disk space must have enough disk space to complete the entire result set, unlike an online index-building operation that can reuse disk space by splitting the operation into multiple cycles.

Estimating a QUERY_MEMORY_LIMIT Value for Queries

To estimate a reasonable QUERY_MEMORY_LIMIT value, use the following procedure:

1. Estimate the number of result rows that will be returned.
2. Determine the row size by adding:
   - The size of the data type for each returned column
   - 10 bytes for overhead
   - For GROUP BY operations, 32 bytes per group
3. Multiply the row size by the number of result rows. This number will allow all processing of intermediate results to occur in memory.

4. Adjust this number based on memory available, complexity of queries, and number of users:

   - On UNIX, determine the maximum program data space that a single process can use. On Windows, determine the amount of physical memory on your computer.

     Never set QUERY_MEMORY_LIMIT to exceed this number.

   - Queries with subqueries or queries that use the GROUP BY, DISTINCT, or ORDER BY clause can benefit from a higher memory limit. If GROUP BY operations involve a large number of groups, setting a higher memory limit can improve performance because more processing can occur in memory.

   - In a multiuser environment, however, lower values for QUERY_MEMORY_LIMIT often work better because they avoid excessive paging that occurs when multiple individual queries consume large amounts of physical memory.

   The value you enter is automatically rounded up to the nearest 8-kilobyte block.

Estimating a MAXSPILLSIZE Value for Queries

To estimate a reasonable MAXSPILLSIZE value for query temporary slices, use the following procedure:

1. Determine the maximum amount of disk space available for query temporary operations and the number of users that will be running queries that spill. Multiple users running large queries will compete for space in the query temporary-space directories. You might want to consider providing separate temporary-space directories for individual users (with SET commands).
2. Choose a MAXSPILLSIZE value that limits the space used by each spill so that all users can be accommodated during peak loads. (The value you enter is automatically rounded up to the nearest 8-kilobyte block.)

Planning for Segmented Storage

After you have planned the database design and estimated the disk storage required for fully loaded and indexed user and system tables, decide whether you want to use named or default storage segments. Depending on the initial space requirements and growth patterns expected for the database, you might choose to assign all user tables and indexes to default segments. The creation and maintenance of default segments is simpler, but they do not have the flexibility offered by named segments. For more information about default and named segments, refer to "Segmented Storage" on page 2-7 and Chapter 9, "Maintaining a Data Warehouse."

Default segments are created when a CREATE TABLE statement containing no explicit segment assignment is issued. Default characteristics of default segments are defined by parameters in the **rbw.config** file. No explicit administrator action is required to create default segments. Files associated with default segments reside in the database directory or in a default directory if one is specified in the **rbw.config** file. A default segment can be modified with an ALTER SEGMENT statement.



Warning: *If your warehouse contains multiple databases and you plan to run offline-load operations on different databases at the same time, do not plan to use a single default directory for all default segments.*

Named segments are explicitly created with a CREATE SEGMENT statement, and the list of files assigned to a named segment is managed with the CREATE SEGMENT and ALTER SEGMENT statements. The files associated with a named segment reside at locations specified when the segment is created or altered.

Determining When to Use Default and Named Segments

Storing tables and indexes in default segments simplifies database creation and administration. However, you must use named segments in the following cases:

- If the estimated size for a single user table or its associated automatic and optional indexes exceeds 2 gigabytes, you might want to place that table in a named segment to distribute database files across multiple file systems.

  A default segment can be larger than 2 gigabytes if you specify a larger size for the DEFAULT_SEGMENT_SIZE parameter in the **rbw.config** file. For more information about configuration file parameters, see Appendix B, "Configuration File."
- If the total estimated size for the database exceeds 4 gigabytes, some tables and indexes must be placed in named segments.
- If the total estimated size for a table or its associated indexes or for the entire database exceeds the space available on the file system where the database directory is to reside, some tables or indexes must be placed in named segments with PSUs in other file systems.

If you expect your database to grow to the point that you will want to distribute data over multiple segments, you probably should use named segments.

However, if you create a table or index in a default segment and later decide you want to add more (named) segments, you can do so after specifying a segmenting column (with an ALTER TABLE statement). Or if you initially choose to use a default segment but later find that a table or index outgrows its default segment, you can use the ALTER SEGMENT statement to move the single PSU to another location or add additional PSUs to the segment.

If you plan to use named segments, keep the following considerations in mind:

- A named segment can contain only one table or one index (or nothing).
- A segment can contain up to 250 PSUs (files). You must decide whether to assign a few large files or more smaller files to the segment. A few large files might require reservation of entire disk partitions whereas use of smaller files might allow fragments of disk space on various file systems to be effectively used for a single table or index. In making this decision, consider that managing and maintaining a few large files is generally easier than managing and maintaining many small files.
- You must also decide how to allow files to grow. For each PSU, you can specify an initial size, which is always reserved at the time the segment is created; a maximum size to which the PSU can grow; and an extend size, which is the increment by which the PSU will grow. Large initial sizes for PSUs require more space to be reserved initially whereas smaller initial sizes are more likely to result in fragmentation.
- Red Brick Decision Server uses disk space in 8-kilobyte blocks. When a maximum file size is specified in a CREATE SEGMENT statement, the space allocated is rounded up to the nearest 8 kilobytes. The first file in a segment is always allocated a minimum of 2 blocks, or 16 kilobytes.
- If your system configuration permits, spread segment files across multiple disk drives and multiple I/O buses to improve access times.

- Segments (of multisegmented tables and indexes) can be taken offline for loading or restore operations and continue to provide limited query access. If this feature is relevant to your site, carefully consider how to segment the data and the locations and sizes of PSUs needed to accommodate the data, keeping in mind the uses you intend to make of offline operations on segments.
- The amount of parallelism used for query processing is limited by the number of PSUs in which the data being queried is stored. For example, if a table contains only a single PSU, no parallel processing occurs for queries that require a relation scan of that table. Therefore, keep in mind the desired degree of parallelism when you define PSUs. For more information about PSUs and parallelism, refer to Chapter 10, "Tuning a Warehouse for Performance."

Considerations for Growing Tables

In some applications, database tables grow over time. Red Brick Decision Server supports the addition of records to existing tables with the incremental load facility of the TMU or with the INSERT statement. By placing large, growing tables in named segments, you have the flexibility to allocate additional storage when and where you need it.

Disk space reserved for growing tables is not actually used until required by incremental row additions, so the space does not need to be available at the time the segment is created.

You might have some experienced and knowledgeable users whom you permit to create *transient tables*, not temporary tables, but real user tables that exist for only a short period of time. These experienced users use transient tables to store intermediate query data or data extracted from production data warehouses. You might not want to give any users permission to create named segments, but you might want to allow these experienced users to create very large, greater than 2-gigabyte, transient tables. In this case, you can set large values for the parameters DEFAULT_SEGMENT_SIZE and DEFAULT_PSU_EXTENDSIZE in the **rbw.config** file. For more information about configuration file parameters, see Appendix B, "Configuration File."

With growing tables, set up a periodic administrative procedure to determine and evaluate current space usage, monitoring table growth and available space by querying the system tables. As tables grow, you can use the ALTER SEGMENT statement to incrementally allocate space as necessary.



Warning: *Because disk space is not actually allocated until it is needed, you might run out of disk space as you add the data even though a sufficient amount is specified in a CREATE SEGMENT statement.*

Effect of Table Growth on STAR Indexes

For tables that will grow, you must also consider how the STAR indexes will grow.

A STAR index relates a referencing table to other referenced tables through FOREIGN KEY clauses. At the time a STAR index is built, certain internal aspects of the index are statically determined, based on the MAXSEGMENTS and MAXROWS PER SEGMENT values specified for the dimension tables referenced by the fact table. You must specify MAXROWS PER SEGMENT when you create dimension (referenced) tables that will participate in a STAR index. Otherwise the CREATE STAR INDEX statement fails.

A STAR index must be large enough to accommodate any rows added to the referenced tables. If it is not large enough, it might become invalid when new rows are added. If a STAR index becomes invalid, it must be either rebuilt using the REORG command of the TMU or dropped and re-created. With frequently updated referenced (dimension) tables or large related referencing (fact) tables, the overhead of performing regular REORG operations can become prohibitive.

If you change the values of MAXSEGMENTS and MAXROWS PER SEGMENT for a growing referenced table, a REORG operation is often needed when records are added to the table or when an ALTER SEGMENT statement is used to expand the segment. If you use accurate values for the MAXSEGMENTS and MAXROWS PER SEGMENT parameters to reserve sufficient space in advance for growing dimension tables, you can avoid the need to alter a segment and perform a REORG operation or at least anticipate it and plan in advance.

Informix strongly recommends that you specify MAXSEGMENTS and MAXROWS PER SEGMENT values for each table. You cannot create a STAR index that references a table unless these parameters have been defined.

Chapter

5

Creating a Database

Overview 5-3

Creating the Database Structure 5-4
- Database Creation Syntax 5-4
- Initializing the Database. 5-6
- Defining a Logical Database Name 5-8
- Changing the DBA Account Password. 5-9

Creating the Database Objects 5-11

Creating Segments 5-12

Creating Tables 5-13
- Setting the MAXSEGMENTS and MAXROWS PER SEGMENT Options 5-14
- Naming Constraints for Primary and Foreign Keys 5-15
- Maintaining Referential Integrity with ON DELETE 5-15

Creating Indexes. 5-16
- INDEX TEMPSPACE Parameters 5-17
- Parallel Indexes. 5-17
- Loading Tables with Indexes 5-18
- STAR Indexes 5-19
- TARGET Indexes 5-19

Creating Views 5-19

Creating and Managing Macros 5-21
- Guidelines for Macro Definitions. 5-22
- Availability and Scope 5-23

In This Chapter

After you design the database schema and plan its implementation, you are ready to create the database system tables and other database objects. This chapter describes the process of creating a database: initializing the database and creating its tables, indexes, and other database objects. This chapter includes the following sections:

- Overview
- Creating the Database Structure
- Creating the Database Objects
- Creating Segments
- Creating Tables
- Creating Indexes
- Creating Views
- Creating and Managing Macros

Overview

Some of the tasks described in this chapter will be performed multiple times over the life of a database as it is revised to accommodate changing user requirements. For example, tables might be added to or removed from a database, and the definition of views, macros, and user privileges might change.

This chapter offers guidelines for creating the various database objects and provides some examples. For a complete description of the SQL syntax, refer to the *SQL Reference Guide*. For a complete example of how to create a database, refer to Appendix A, "Example: Building a Database."

To create a database

1. Create the database structure using the **rb_creator** utility on UNIX or the **dbcreate** utility on Windows.
2. Create the database objects, including segments, user tables, indexes, synonyms, views, and macros, based on the logical schema you defined and on the physical implementation you have chosen.

After you create the database, you must provide user access, as described in Chapter 7, "Providing Database Access and Security," and load data into the database with the Table Management Utility (TMU), as described in the *Table Management Utility Reference Guide*.

Creating the Database Structure

To create the database structure, use the **rb_creator** utility on UNIX or the **dbcreate** utility on Windows to initialize the database and create the system tables. Then define a logical name for the database in the **rbw.config** file. Also change the default password for the database administrative account. Each of these tasks is described in this section.

By default, databases are created in the locale specified when Red Brick Decision Server is installed. If you want a different locale, you must choose it from the locales that are available. For information about specifying a locale, refer to "Server Locale" on page 2-27. A list of locales is provided in the **locales.pdf** file in the **relnotes** directory on the Red Brick Decision Server product CD-ROM.

Database Creation Syntax

Create a database with the **rb_creator** utility on UNIX or the **dbcreate** utility on Windows.



Important: *The **RB_CONFIG** environment variable must be set to use the **rb_creator** or **dbcreate** utilities. For more information on **RB_CONFIG**, see "Environment Variables" on page 2-25.*

UNIX

To create a database on UNIX:

```
rb_creator dirname ["language_territory.codeset@sort"]
```

dirname	The database directory
"*language_territory.codeset@sort*"	Optional. The locale for the database if the locale differs from the Red Brick Decision Server locale (set with the NLS_LOCALE LOCALE parameter in the **rbw.config** file)

♦

WIN NT/2000

To create a database on Windows:

```
dbcreate -d dirname [-l logical database name] [-a] [-u username]
[-p password] [-n "language_territory.codeset@sort"]
```

-d *dirname*	The database directory
-l *logical database name*	Optional. The logical name of the database to be added to the **rbw.config** file
-a	Optional. For creating the administration database
-u *username*	Optional. The user name (Default is SYSTEM)
-p *password*	Optional. The password (Default is MANAGER)
-n "*language_territory.codeset@sort*"	Optional. The locale for the database if the locale differs from the Red Brick Decision Server locale (set with the NLS_LOCALE LOCALE parameter in the **rbw.config** file)

♦

For instructions on deleting a database, see "Deleting a Database" on page 9-68.

Initializing the Database

Initialize a new database with the **rb_creator** utility on UNIX or the **dbcreate** utility on Windows.

UNIX

To initialize a new database

1. Log in as the **redbrick** user and change to the parent directory for the location in which you want to create the database.
2. Create the database directory.

   ```
   $ mkdir dirname
   ```

   dirname The pathname to the directory, such as **/disk1/*database***

   The **redbrick** user must have access permissions to create this directory, and the directory must be empty.
3. Verify that permissions are set correctly for this directory by entering:

   ```
   $ ls -l
   ```

   Permissions on the directory should be as follows:

   ```
   redbrick:   rwx   (read, write, execute)
   group:      ---   (none)
   other:      ---   (none)
   ```

   If permissions do not match these settings, verify that the **umask** setting for the **redbrick** user is correct (`077`), and use the system **chmod** command to set the permissions correctly.
4. Create the database using the **rb_creator** utility. For the syntax, see "Database Creation Syntax" on page 5-4.

   If the directory you specify does not designate an empty directory or if you (as the **redbrick** user) do not have sufficient write privileges, **rb_creator** exits with an error message and does not create a new database. Otherwise, **rb_creator** initializes the database by creating the database system files listed in Figure 5-1 on page 5-8.

♦

WIN NT/2000

To initialize a new database

1. Log in as the **redbrick** user and change to the parent directory for the location in which you want to create the database.
2. Create the database directory.

   ```
   c:\> mkdir dirname
   ```

   dirname The pathname to the directory, such as **c:\disk1*database***

3. Create the database using the **dbcreate** utility. For the syntax, see "Database Creation Syntax" on page 5-4.

   Running **dbcreate** initializes the database by creating the database system files listed in Figure 5-1 on page 5-8.

When you use the **dbcreate** utility to initialize a database, you must fully qualify the pathname for the database directory by including the drive letter. For example:

```
c:\> dbcreate -create -l AROMA -d c:\disk1\new_db
```

The database pathname is added to the **rbw.config** file automatically. If you edit the database section of this file, be sure that the pathname for each database is fully qualified.

If two logical database names in the **rbw.config** file refer to the same physical database, the pathnames must be identical for the two entries.

♦

When you run **rb_creator** on UNIX or **dbcreate** on Windows, the database server initializes the database by creating the following database system files.

Figure 5-1
Database System Files

System File	Contents
RB_DEFAULT_IDX	System tables
RB_DEFAULT_LOCKS	System information for database and table locks
RB_DEFAULT_INDEXES	System information about indexes
RB_DEFAULT_SEGMENTS	System information about segments
RB_DEFAULT_TABLES	System information about tables



Tip: *The database system file* ***RB_DEFAULT_LOADINFO,*** *which contains information about load operations, is not created until a load operation occurs.*

Defining a Logical Database Name

When you create a new database, assign it a logical database name in the **rbw.config** file. Users then access the database by its logical database name. The logical database name can be up to 128 characters in length.

To assign a logical database name

1. Open the **rbw.config** file for editing. The section to be edited is titled `Logical database name mappings`. The file at your site might contain other database names but looks similar to this:

   ```
   # Logical database name mappings
   #
   DB AROMA <dirname>
   ```

   The **<*dirname*>** variable is equivalent to the full pathname; for instance, **/disk1/aroma/db** on UNIX or **c:\disk1\aroma\db** on Windows.

2. Insert a line with the following form after the `Logical database name mappings` header:

   ```
   DB database_name dirname
   ```

database_name	Logical database name for new database; case insensitive. (Users can access using uppercase, lowercase, or a combination of both cases.)
dirname	Full pathname to the database directory. (On UNIX, this information is case sensitive.)

WIN NT/2000

You can also use the **-l** command-line option of **dbcreate** to automatically add the logical database name to the **rbw.config** file when you create the database. ♦

Example

If you want to add a new database named **NEW_DB**, which is in the <*pathname*> directory (for instance, **/disk1/new_db** on UNIX or **c:\disk1\new_db** on Windows), add the following line to the **rbw.config** file:

```
DB  NEW_DB <pathname>
```

If the file previously contained an entry only for the **Aroma** database, it would now look like this.

```
# Logical database name mappings
#
DB AROMA <pathname>
DB NEW_DB <pathname>
```

Changing the DBA Account Password

At the system level, the new database files are owned by the **redbrick** user. At the database level, each new database is created with a single database user account named **system**, with the default password **manager**. This user account is a member of the DBA system role, with the authorization and privileges of that role. Change the default password from **manager** to a secure password immediately after creating the new database.

To change the default password

1. To invoke the RISQL Entry Tool, enter the following command at the prompt:

   ```
   risql -d logical_database_name system manager
   ```

   logical_database_name Name of the new database.

2. Change the password.

   ```
   RISQL> grant connect to system with new_password;
   ```

A password can be any valid database identifier or string literal, as defined in the *SQL Reference Guide*.

WIN NT/2000

You can also use **dbcreate** to change the password of the **system** account by specifying the **-u** and **-p** parameters when you create the database. ♦

Example

This example illustrates how to change the password for the user **system** from the default password, **manager**, to the new password, **mysecret**.

UNIX

```
$ risql -d new_db system manager
(C)Copyright 1991-2000, Informix Software, Inc.
All rights reserved
Version 6.10.0
RISQL> grant connect to system with mysecret;
RISQL> quit;
$
```

♦

WIN NT/2000

```
c:\> risql -d new_db system manager
(C)Copyright 1991-2000, Informix Software, Inc.,
All rights reserved
Version 6.10.0
RISQL> grant connect to system with mysecret;
RISQL> quit;
c:\>
```

♦

Creating the Database Objects

After the database has been created with **rb_creator** (UNIX) or **dbcreate** (Windows), you can create the objects (segments, user tables, indexes, and so on) that make up the database schema. You create each type of object with an SQL CREATE statement.

To enter complex CREATE statements, write them in a text file to use as an input script for the RISQL Entry Tool or RISQL Reporter. You can also enter simple table organizations interactively from the RISQL Entry Tool command line or with any tool that supports direct entry of SQL. The Manage Tables function of the Administrator tool offers an easy method to create tables and can write the SQL statements for you.

For information about using script files, refer to the *RISQL Entry Tool and RISQL Reporter User's Guide*. For information about writing CREATE TABLE statements, refer to "Creating Tables" on page 5-13.

To create database objects

1. If you are using named segments for some or all of the tables or indexes in the database, write the necessary CREATE SEGMENT statements.
2. Write CREATE TABLE statements for the dimension (referenced) tables. Use the MAXROWS PER SEGMENT and MAXSEGMENTS parameters to specify the expected number of rows for each table.
3. Write CREATE TABLE statements for the fact (referencing) tables.
4. Write CREATE INDEX statements for any STAR indexes, additional B-TREE indexes, or TARGET indexes that you have selected as described in "Creating Indexes" on page 5-16.

   If you have written the CREATE statements in a script file, process them by invoking the RISQL Entry Tool and reading the script file.
5. Write CREATE VIEW statements to provide the convenience and security afforded by views and to make the system aware of aggregate tables.

6. Write any desired CREATE MACRO statements to simplify repetitive query components or to share procedures.
7. Create precomputed views and hierarchies to optimize aggregate query performance, as explained in the *Informix Vista User's Guide*.

For a complete description of the SQL syntax, refer to the *SQL Reference Guide*.

Creating Segments

The CREATE SEGMENT statement creates a storage segment consisting of one or more PSUs that together will contain a table or index. If you plan to use named segments, you must define them before you create the tables and indexes. Segments can be modified as needed with an ALTER SEGMENT statement. For information about modifying segments, refer to "Altering Segments" on page 9-29.

The following rules about segments apply:

- Named segments are created with the CREATE SEGMENT statement. Default segments are created automatically when named segments are not specified in CREATE TABLE or CREATE INDEX statements. The characteristics of default segments are defined in parameters of the **rbw.config** file.
- The default location for default segments is the database directory. If you have multiple databases and want to specify a different default directory for one or more of the databases, use a SET command to define these locations rather than an **rbw.config** file entry for DEFAULT_DATA_SEGMENT or DEFAULT_INDEX_SEGMENT.
- Named segments for all tables and their primary key indexes are assigned with the CREATE TABLE statement. Named segments for all optional indexes are assigned with the CREATE INDEX statement.
- Any table can reside in multiple segments, with data distributed by the data values or by a hashing algorithm.

- Each index on a table can reside in multiple segments. For primary indexes, the index entries can be distributed among segments in the same way as the indexed data (SEGMENT LIKE DATA) or by ranges that are independent of the data distribution. For STAR indexes, the index entries can be distributed sequentially among segments or distributed by references to internal storage locations (SEGMENT BY REFERENCES OF).
- Segments can span multiple PSUs.

Disk space is first allocated to a segment based on the INITSIZE of the first PSU in the segment (according to the PSU sequence ID in the **RBW_SEGMENTS** table). Additional space is then allocated as data is stored in the PSU. Segments with large INITSIZE values take longer to create (because the INITSIZE space is being allocated) but are faster to load than segments with smaller INITSIZE values.

Creating Tables

Each user table is defined in a CREATE TABLE statement with a name, a description of the column (including name, data type, and special instructions), and optional primary key definition, foreign key definitions, referential integrity action, segment identifiers, and the values for MAXSEGMENTS and MAXROWS PER SEGMENT (for tables that will participate in STAR indexes).

Tables can be modified as needed with an ALTER TABLE statement. For information about modifying tables, refer to "Altering Tables" on page 9-39. For more information on data types, refer to the *SQL Reference Guide*. For information on specifying fill factors for VARCHAR columns, refer to and to the *SQL Reference Guide*.

The following rules apply to creating tables:

- A table containing a foreign key reference from another table must be created before the table that references it.
- An outboard table must be created before any tables that reference it.

Setting the MAXSEGMENTS and MAXROWS PER SEGMENT Options

Informix recommends that you specify the expected total number of segments and total number of rows in a segment for a table as the MAXSEGMENTS and MAXROWS PER SEGMENT value for that table. These values are used to build a STAR index that can accommodate the expected growth of the dimension tables that participate in it. If a MAXROWS PER SEGMENT value is not provided on a referenced table, the STAR index creation fails.

Specifying MAXSEGMENTS and MAXROWS PER SEGMENT values larger than the current (or expected) size of the dimension tables results in a STAR index that is larger than necessary. You can perform the calculations described in "Estimating the size of indexes" on page 4-22 using various values for MAXSEGMENTS and MAXROWS PER SEGMENT to see what effect this parameter has on the STAR index size.

The MAXSEGMENTS and MAXROWS PER SEGMENT values are also needed to validate STAR index segmentation and to use the TMU Automatic Row Generation option to maintain referential integrity.

The MAXROWS PER SEGMENT limit is affected by the fill factor settings for VARCHAR columns. A fill factor setting that is too low might cause this limit to be reached prematurely. If you receive an error message while inserting rows into a table containing VARCHAR columns that the maximum number or rows per segment has been reached, verify that the fill factor setting is correct before you adjust the MAXROWS PER SEGMENT value. For more information, refer to "Setting the VARCHAR Column Fill Factor" on page 10-28.

Naming Constraints for Primary and Foreign Keys

A constraint name is a logical name associated with a primary key or a foreign key and is defined with the CONSTRAINT keyword in a CREATE TABLE or ALTER TABLE statement. For multicolumn foreign key references, constraint names are required if the multicolumn foreign key is referenced in a STAR index. Although constraint names are optional for single-column foreign keys and for primary keys, having meaningful constraint names can make your CREATE TABLE statements more understandable to other people. Constraint names are identified in the **RBW_RELATIONSHIPS** and **RBW_CONSTRAINTS** system tables. If constraint names are not supplied, they are given default names.

For more information on the CONSTRAINT keyword of the CREATE TABLE statement, refer to the *SQL Reference Guide*.

Maintaining Referential Integrity with ON DELETE

To maintain referential integrity during delete operations, you must consider both table creation and delete operations on related tables. Referential integrity is the relational property that each foreign key value in a table exists as a primary key value in the referenced table.

The ON DELETE clause in the FOREIGN KEY clause of the CREATE TABLE statement specifies how referential integrity is maintained during delete operations. The ON DELETE clause has the following options.

Option	Description
CASCADE	If a to-be-deleted row is referenced by a row or rows in another table, both that row and the referencing rows are deleted. The delete cascades through all affected tables to maintain referential integrity.
NO ACTION	If a to-be-deleted row is referenced by a row or rows in another table, neither that row nor the referencing row is deleted from either table.This type of delete is also called a restricted delete. However, a row that is not referenced by another row will be deleted. (That behavior is the default if ON DELETE is omitted.)

The delete setting is stored in the **DELACTION** column of the system table **RBW_RELATIONSHIPS**.

A delete operation cannot perform both cascaded and restricted deletes. A NO ACTION reference anywhere in the complete family of a table (as defined in "Delete Operations and Cascaded Deletes" on page 2-41.)overrides any CASCADE references in the family on a row-by-row basis. The behavior is as if all references were NO ACTION.

For a specific delete operation, you can override the ON DELETE action specified when the table was created by using the OVERRIDE REFCHECK clause in the DELETE statement.



Warning: *Although performance during delete operations is better when referential integrity is not checked, use OVERRIDE REFCHECK only when you know deletions will not violate referential integrity or when you plan to ensure referential integrity by performing a REORG operation on the referenced table after the delete operation.*

You can also change the ON DELETE action associated with the foreign key for all future delete operations by using the ALTER TABLE statement to alter the column.

To ensure that referential integrity is maintained in the most appropriate manner for your database, be sure that you understand both actions and select the most effective combination for all tables in your database.

Creating Indexes

A B-TREE index is automatically created on the primary key columns of a table at table creation. You can create optional indexes to improve query performance. Red Brick Decision Server has three types of index technologies: STAR indexes, TARGET indexes, and B-TREE indexes. For guidelines about when to create additional indexes and when to use the various types of indexes, refer to "Determining When to Create Additional Indexes" on page 4-4. For further information on the CREATE INDEX statement, refer to the *SQL Reference Guide*.

While an index is being built, the table can be read by other users but not written. If an index is being built, database backup operations will not back up that index and will issue a warning message to that effect.

You can create an index before data is loaded into the table, in which case the index is built as the data is loaded, or you can create it later on a populated table. You can drop any index at any time.

INDEX TEMPSPACE Parameters

Set the INDEX TEMPSPACE DIRECTORIES location to large empty disk partitions to minimize the chances of running out of temporary space for offline load and index-building operations. With multiple databases, this parameter should specify a different location for each database.

The value specified for INDEX TEMPSPACE THRESHOLD determines the size of the memory work area. For LOAD, REORG, and offline LOAD procedures, the recommended settings work well. Customization can be helpful on some systems.

For CREATE INDEX operations that create multiple indexes in parallel, the amount specified by this parameter is distributed among all indexes. Therefore, the operating system might run out of memory and swap space. If you encounter out-of-memory errors when building indexes in parallel, try building fewer indexes in parallel, reducing the INDEX TEMPSPACE THRESHOLD value, or increasing the swap space.

The goal in selecting the INDEX TEMPSPACE THRESHOLD value is to select a large value relative to the system on which Red Brick Decision Server is running, but not so large a value that errors occur. Selecting too large a value might cause the operation to fail with an out-of-memory error. Selecting too small a value might cause poor performance.

For more information about INDEX TEMPSPACE parameters, refer to "Estimating Temporary Space Values for Index-Building Operations" on page 4-37.

Parallel Indexes

UNIX

The CREATE INDEX statement provides a parallel-index-creation capability that allows multiple indexes to be built simultaneously on multiprocessor hardware platforms. Although designed to improve performance on multiprocessor systems, the ability to create multiple indexes with a single statement is convenient on any system.

Creating multiple indexes in parallel on a table with a single CREATE INDEX statement is quicker on multiprocessor systems and often more convenient than creating each index with a separate CREATE statement.

The ON ERROR clause specifies what happens if an error occurs while multiple indexes are being built. You can specify that all index building stop (ABORT) or that building continue for other indexes (CONTINUE) not affected by the error.

After building multiple indexes in parallel with a single CREATE INDEX statement, verify that all indexes were created successfully by checking the **DATETIME** column in the **RBW_TABLES** or **RBW_INDEXES** system table. The **DATETIME** column indicates NULL for each index under construction and the time and date of completion for each index when it is complete. If the database server was not able to successfully complete an index, it deletes the index entry from **RBW_INDEXES** automatically in many cases. However, if any index indicates NULL for the DATETIME value after the CREATE statement completes, manually remove that index entry from **RBW_INDEXES** with the DROP INDEX statement. ♦

WIN NT/2000

On Windows, you cannot use the CREATE INDEX statement to create indexes in parallel. You can achieve the same effect by deferring the creation of the index with a CREATE INDEX DEFERRED statement followed by a REORG command. For syntax of the REORG command, refer to the *Table Management Utility Reference Guide*. ♦

Loading Tables with Indexes

For large dimension tables (more than 10,000 rows) with user-defined indexes, creating the indexes after the load process has completed is usually more efficient than creating them before the load process.

For dimension tables with multiple indexes, it is usually faster to drop indexes before loading data and then re-create the indexes using the parallel indexes feature after the load operation is complete. This method also has the advantage that the table is available for query while the indexes are being built, although with lower query performance.

STAR Indexes

You can create one or more STAR indexes on any table that has foreign key references. If the table has a more than one foreign key, you can create a STAR index on any combination or subset of those foreign keys. A STAR index can greatly improve query performance when you are using a star schema design. For more information about STAR indexes, refer to "STAR Indexes" on page 4-6.

TARGET Indexes

You can create a TARGET index on any column of a table. TARGET indexes can improve performance on queries involving columns that have weakly selective constraints. For example, if a column has five possible values and your query constrains on one of those five values, a TARGET index can help you retrieve this information very efficiently. Furthermore, TARGET indexes provide a fast way of counting the number of occurrences of a weakly selective constraint. For more information about TARGET indexes, refer to "TARGET Indexes" on page 4-10.

Creating Views

A view consists of selected columns and rows from tables within a database. Views allow simplification of queries and hiding of data. For example, if a frequent query references only certain columns or rows, define a view containing only those columns and rows. If some tables contain confidential information, define views that include or exclude that data and grant access on those views as appropriate.

A view can be created or dropped at any time, regardless of whether the tables referenced by the view contain data. Any table referenced by a view must exist at the time the view is created.

An optimized internal form of the view is built when it is defined. Therefore, if a view includes a macro or a SELECT * ... statement, the macro or SELECT statement is expanded when the view is defined. Subsequent changes to the macro or columns that are added to the table are not reflected in the view. A view itself cannot be updated. That is, you cannot insert, update, or delete rows in a view. Views reflect changes to the data in the tables on which they are based.

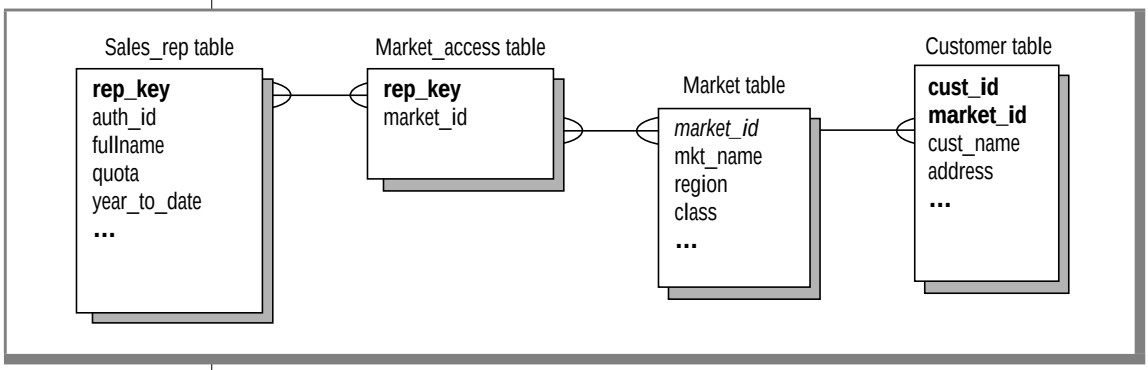
If the text defining a view exceeds 256 bytes, multiple rows are inserted in the system table **RBW_VIEWTEXT**. In views that include a macro, only the macro name, not the expanded macro, is included in the character count.

You can also create *precomputed views* and *hierarchies* to automatically rewrite and optimize aggregate queries. For information about the query rewrite system, refer to the *Informix Vista User's Guide*.

Example

Assume you want to limit access of the **Customer** table by sales representatives to those customers in their own market areas, with managers having access to multiple market areas. The relevant portion of the schema contains the following tables and columns.

Figure 5-2



You can accomplish the desired restriction by creating a view for each user.

```
create view user1_list
      as select * from customer
      where market_id in
         (select market_id from market_access, sales_rep
            where sales_rep.auth_id = 'user1'
            and sales_rep.repkey = market_access.rep_key);
```

Then grant SELECT privileges to each user for that user's view. If you have many users, however, this approach requires that you create and maintain many individual views.

A more general view that uses the SQL CURRENT_USER (or USER) function to restrict access is easier to maintain.

```
create view cust_list
      as select * from customer
      where market_id in
         (select market_id from market_access, sales_rep
            where sales_rep.auth_id = CURRENT_USER
            and sales_rep.repkey = market_access.rep_key) ;
```

Then grant SELECT privileges for public access on the view **cust_list**.

```
grant select on cust_list to public ;
```

Sales representatives can then query the database as follows to see customers in their own market areas:

```
select * from cust_list ;
```

The only customers displayed are those in markets that the sales representative has permission to access, as defined by the **Market_Access** table.

Creating and Managing Macros

The definition and use of macros is optional, but they can make writing SQL statements easier for both the database administrator and the general user. A macro can be defined to shorten lengthy character strings or to simplify complex queries. A macro can also be generalized with parameters or nested within another macro. Each person who writes SQL statements should analyze the statements for patterns and similarities that could be simplified or reduced by the use of macros.

Guidelines for Macro Definitions

Some general guidelines for macro definitions follow:

- The definition string within each CREATE MACRO statement must be less than 1024 characters.
- A macro definition can include other macros unless the definition is circular; that is, macro **a** includes macro **b**, and macro **b** includes macro **a**. In such a case, an error message is issued when the macro is expanded during execution.
- A macro definition can include parameters, which generalize the definition.
- A macro definition can include a category and a descriptive comment.
- Macro definitions are stored in the TEXT column of the **RBW_MACROS** system table. You can query the table to verify the definition of a macro. You can also use the EXPAND statement to see how a macro is expanded, including any parameter values.

The CATEGORY and COMMENT values are stored in the corresponding columns of the **RBW_MACROS** system table. The CATEGORY value is not used by Red Brick Decision Server but is intended for use with application tools supplied by other vendors. For example, the category might be used to specify how the macro syntax fits into the full SQL statement syntax. The COMMENT value can store a descriptive comment about the macro or other information for use with application tools.

For a complete definition of the CREATE MACRO statement, refer to the *SQL Reference Guide*.

Availability and Scope

The way in which a macro is defined determines its availability and scope. Red Brick Decision Server supports the following types of macro definitions:

- Public macros, which are available to all database users. These macros can be defined only by users with DBA authorization and are stored in the **RBW_MACROS** system table.
- Private macros, which can be used only by the creator of the macro. These macros can be defined only by users with DBA or RESOURCE authorization and are also stored in the **RBW_MACROS** system table.
- Temporary macros, which can be used only by the creator of the macro. These macros can be defined by anyone and are available only during the session in which they are defined. These macros can be stored in **.rbwrc** files that are read in automatically each time a session is started. For tools that start and end a new session for each query, temporary macros are of limited use unless the CREATE MACRO statements are stored in **.rbwrc** files.

The following table defines the macro definition rules and scope.

Macro Type	Required Authorization	For User	Syntax
Public	DBA (CREATE ANY task)	All	CREATE PUBLIC MACRO …
Private	DBA or RESOURCE (CREATE ANY or CREATE OWN task)	Creator	CREATE MACRO…
Temporary	CONNECT (any user)	Creator	CREATE TEMPORARY MACRO…

The way in which macros are handled varies among the individual tools used with Red Brick Decision Server. For information about macro use by a particular tool, refer to specific Informix or vendor information about that tool.

Macro references are resolved as follows:

- If a temporary macro with the name exists, it is used.
- If no temporary macro exists and if a private macro with the name exists, the private macro is used.
- If no temporary or private macro exists and a public macro with the name exists, the public macro is used.

Examples

This example illustrates how a frequently used block of SQL code can be replaced with a simple macro. A macro named **std_select** is defined to shorten an often-repeated select list, FROM clause, and series of WHERE clause constraints.

```
create macro std_select as
   date_col, product, city, dollars
   from period, product, market, sales
   where period.perkey = sales.perkey
   and product.prodkey = sales.prodkey
   and market.mktkey = sales.mktkey ;
```

To use the following macro in a SELECT statement that finds sales data for San Jose in 1998:

```
select std_select and year = 1998 and city ='San Jose';
```

This example illustrates how frequently used blocks of similar instructions can be replaced with a parameterized macro. A macro named **std_constraint** is defined with two parameters to provide a general-purpose constraint for a frequently asked query.

```
create macro std_constraint (yr, cty) as
   year = yr and city = cty;
```



Tip: *The parameter name should not match any text in the definition that you do not want replaced with the parameter value. For example, the following definition, while legal, defines a constraint that is always satisfied:*

```
create macro std_constraint (year, city) as
   year = year and city = city;
```

To verify that a macro is defined correctly, check the definition by querying the **RBW_MACROS** system table. The macro identifier is stored as uppercase letters.

```
select name, text from rbw_macros
   where name = 'STD_CONSTRAINT';
NAME                    TEXT
STD_CONSTRAINT          YEAR=%1 AND CITY=%2
```

To verify that a macro is expanded correctly, particularly when parameters are used, check the macro expansion with the EXPAND statement.

```
expand std_constraint(1998, 'San Jose');
STATEMENT
YEAR=1998 AND CITY='San Jose';
```

To use the parameterized macro in a SELECT statement to find 1998 sales data for both San Jose and Miami:

```
select std_select and std_constraint (1998,'San Jose');
...
select std_select and std_constraint (1998,'Miami');
...
```

The following macro is defined to provide a single statement for these frequently asked, similar queries. It contains two embedded, or nested, macros.

```
create macro std_query (yr, cty) as
   select std_select and std_constraint (yr, cty);
```

The following macro allows you to use a short statement to find the desired sales data:

```
std_query (1998,'San Jose');
std_query (1998,'Miami');
```

For more examples of macro definitions, refer to the *SQL Self-Study Guide*.

Chapter

6

Working with a Versioned Database

Determining Whether You Need Versioning 6-4
- Load Window 6-4
- Increased Recoverability 6-4
- Load with Periodic Commit 6-5
- Providing Load Recoverability 6-6
- Versioning with Vista. 6-6
- Dimension Table Cleaning 6-7
- Costs of the Version Log. 6-7

Loading Data into Versioned Databases. 6-8

Understanding the Version Log 6-10
- Structure of the Version Log 6-11
- Versioned DELETE Operations 6-13

Understanding Frozen Versions 6-13

Loading Data with the High Recoverability Option. 6-16

Controlling Versioning 6-16
- Creating the Version Log 6-18
- Cleaning the Version Log 6-19
- Dropping the Version Log and Adding Space 6-20
- Controlling Frozen Versions 6-21
  - Freezing the Database 6-21
  - Overriding Frozen Versions 6-22
  - Unfreezing the Database 6-22
  - Unloading Data with Frozen Versions 6-22
  - Using Frozen Versions and High Recoverability 6-22

Maintaining a Versioned Database 6-23
Monitoring the Version Log 6-23
Backup and Recovery 6-24

Controlling the Vacuum Cleaner 6-25

Example: Creating a Versioned Aroma Database 6-26

In This Chapter

Versioned databases allow continuous availability, providing the database administrator the capability to load or update the database at the same time it is being queried by users and without impacting query performance. Each transaction creates a new revision, or version, which is assigned a read revision number when the previous transaction commits.

Use versioning to perform large loads during production hours, to facilitate recovery in the event of an abort during a large load, or to periodically refresh the database during the day. You can choose to provide the latest version of the database to user queries or to provide the same version to all queries by using the frozen version mode, thus ensuring consistent results in data analysis while the revision is being loaded and verified.

This chapter covers the following topics:

- Determining Whether You Need Versioning
- Loading Data into Versioned Databases
- Understanding the Version Log
- Understanding Frozen Versions
- Loading Data with the High Recoverability Option
- Controlling Versioning
- Maintaining a Versioned Database
- Controlling the Vacuum Cleaner
- Example: Creating a Versioned Aroma Database

Determining Whether You Need Versioning

Versioning provides the benefits of an increased load window, an increased level of database recoverability, and the ability to add or modify small amounts of data during production hours. If these factors are not important to the operation of your database, you probably do not need to use versioning. System costs associated with versioning relate to the storage space and maintenance of the version log. Also consider the effects on load performance when determining whether to use versioning.

Load Window

The load window is the period when the database is not available for query operations. When you perform a load operation, the database server blocks read access to that table for the duration of the update. If your load window, as determined by user requirements, is shorter than the time needed for database maintenance, versioning might be appropriate for your system.You can use versioning to decrease or eliminate the period of time when the database is unavailable. Versioning can be particularly useful for large load operations.

Your user community will tell you how important availability is. If you have users across several time zones, the time difference might mean that the database needs to be available nearly all the time for all users to have access during their business hours. In this case, versioning enables you to maintain availability. If only a small number of users need access only between the hours of 9:00 A.M. and 5:00 P.M., you might not want to use versioning but instead perform database maintenance at night with the database unavailable to users.

Increased Recoverability

When you run a transaction with versioning enabled, existing blocks in the database are not directly modified. Modified blocks are written to the version log, not to the database files. This arrangement provides complete recoverability to the point when the transaction began because none of the database files are changed. The old revision of the database is completely consistent during the transaction.

For example, suppose you are loading data with versioning enabled and the power fails in the middle of the operation. When the system comes back up, the database is still operational and in the same state as before the operation began. Any transactions that were in process during the power failure are aborted, and the database is automatically returned to a consistent state. You can then start the load operation again. If this situation occurs without versioning, the database might be left in an inconsistent state, requiring some recovery action to make it operational.

Load with Periodic Commit

You might find it useful to load data with a periodic commit interval. Loading incrementally in this manner can reduce the amount of data that you must reload in the event of an abort in the middle of a large load and can also be used to build trickle-feed applications.

Trickle-feed applications are useful if you have a steady stream of data, such as a direct feed from an OLTP system, to load into a database, and you want that data to become visible to users without having to stop and restart the load. Users can then analyze nearly real-time data.

For example, if a database stores information on stock ticker activity, you can load new data in small batches at set intervals of 15 minutes. Without versioning, you would have to disallow queries during that time, and the database would be unavailable during much of the time the stock market is open.

In the default mode, a versioned transaction completes an entire load operation before the commit operation occurs. To build a trickle-feed application, specify more frequent intervals at which a load operation commits with commit interval settings. These intervals can be set to commit after a specified number of rows have been loaded (SET TMU COMMIT RECORD INTERVAL), after a specified time period has elapsed (SET TMU COMMIT TIME INTERVAL), or both.

You can also use the commit interval as a checkpoint mechanism for normal versioned loads. If the load operation fails during the next interval, it returns to the latest committed revision. For example, if you are loading 1,000,000 rows with the TMU COMMIT RECORD INTERVAL set to 100,000 records and the load fails after loading 999,000 records, the load is still complete to the last commit at 900,000 records. If you were performing a normal versioned load that failed in such a manner, that transaction would return to the state before any records were loaded.

For more information on the TMU COMMIT command, refer to the *Table Management Utility Reference Guide*.

Providing Load Recoverability

You can use the version log as a traditional write-ahead log to provide a high level of recoverability: for example, to recover from fatal failures such as running out of disk space. Although they use exactly the same mechanism, the high recoverability option is more predictable than general versioning, both in its use of version log resources and in the case of a problem with the version log itself.

For more information about loading with the high recoverability option, see "Loading Data with the High Recoverability Option" on page 6-16.

Versioning with Vista

If you are using the Vista query rewrite and aggregate maintenance features, you can use versioning to greater advantage. The validity of precomputed views used for aggregate query rewrites is subject to the same rules in both versioned and non-versioned databases. However, by using a frozen query revision, you can modify detail tables without invalidating the associated precomputed views currently being used to rewrite users' queries.

Versioning can also safeguard the consistency of detail and aggregate table data during loads that perform aggregate maintenance. When versioning is in effect and aggregate maintenance fails for some reason, you have two choices of error handling: invalidate the precomputed views in question or roll back all updates to both detail and aggregate tables to make sure they are synchronized. These choices are not feasible in nonversioned databases.

For more information about the query rewrite system and aggregate maintenance, refer to the *Informix Vista User's Guide*.

Dimension Table Cleaning

Versioning allows cleaning of data in dimension tables during production hours. Dimension table cleaning might include deleting unused rows, changing product descriptions or customer addresses, or adding rows rejected by referential integrity checks during the initial load. Versioning allows this data to be updated during production hours, allowing for a more current set of data for the users.

Costs of the Version Log

The costs of the version log are relatively small. They include the following:

- Disk space

  To ensure good query performance, the version log should reside on its own storage subsystem, preferably with several dedicated disk drives. For information on configuring and sizing the version log, refer to "Creating the Version Log" on page 6-18.

- Additional I/O

  When you modify a versioned database, all of the changed blocks are first written to the version log. They are eventually rewritten back to the database files, increasing the I/O. Entirely new blocks, as opposed to modified blocks, are written to the database directly, not to the version log. Versioned transactions that create new blocks incur little additional I/O cost.

- The cost to administer the version log

  Although it is not complex, it does add to the overall complexity of your system.

The cost of versioning is low for load operations that add new table and index data in new segments. These operations change few existing blocks in the database, so few blocks are written to the version log. New data goes directly to the table and index PSUs in the database. The version log is most efficient when the number of blocks written to the version log is relatively small. The more blocks that are written to the version log, the more blocks will eventually be rewritten to the database files, thus increasing I/O.

Loading Data into Versioned Databases

Versioning does not directly affect TMU load performance although possible I/O bottlenecks in accessing the version log after a versioned load might affect system performance. Bottlenecks might occur because queries are accessing data from the version log rather than from the database files at the same time as the vacuum cleaner is reading the version log and users might be modifying the database (writing to the version log).

The main factor affecting performance after a versioned TMU load is the number of existing database blocks that change. If blocks are primarily *new* and few are *changed*, the performance impact is negligible.

Because the TMU updates everything in the database necessary to keep it consistent, consider all of the following items when you analyze the number of changed blocks from your TMU operation:

- Existing rows in tables that change
- Any changes to referenced tables
- Any changes to indexes affected by the operation
- Any changes to maintained aggregate tables and their indexes

Example 1

Consider a simple star schema with a single fact table that references three dimension tables, as in the following figure.

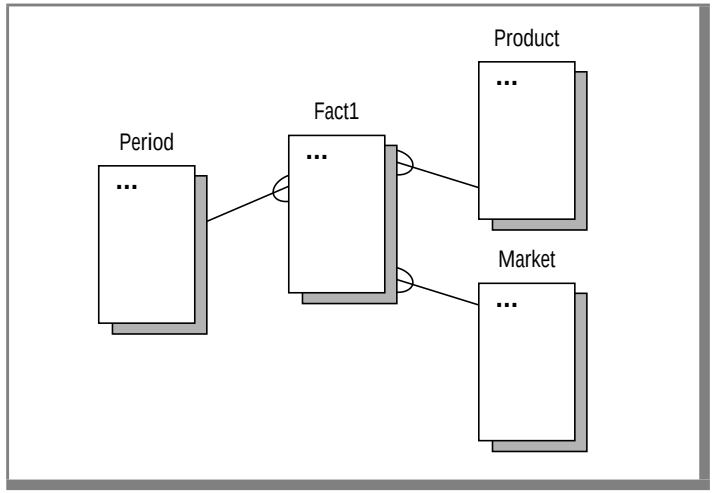


Figure 6-1 Simple Star Schema

Assume that the fact table (**Fact1**) is segmented by month. (These values come from the **Period** table.) Assume also that a single STAR index is created on the three foreign keys of the **Fact1** table and that the index is also segmented by month. The only other indexes in this database are the primary key B-TREE indexes of the three dimension tables.

Because of the segmentation of the fact table and the STAR index, when a new month of data is loaded into this database, all the table data goes into a new **Fact1** table segment and all of the corresponding index data goes into a new STAR index segment. Therefore, there is *no changed data* when a new month of data is loaded.

This scenario creates no increase in I/O to the version log when data is loaded in versioned mode because no table or index data is written to the version log. The data is all written directly to the database files because it is new. Only changed blocks are written to the version log.

Example 2

Consider the same scenario as in the previous example, but with three TARGET indexes on the fact table: one for each foreign key column. Unlike the STAR index, these TARGET indexes are not segmented like the data and therefore contain data for all time periods in the database. When you load the new month of data, many blocks in the TARGET indexes change. Each of those changed blocks is written to the version log. Because of the increased I/O to the version log files, this operation might affect performance after the transaction is committed.

Weigh this cost with the benefits of performing a versioned load. If your users demand 24 hour availability, 7 days a week, it might be worth the extra overhead. However, if no one uses the system at night and the LOAD operations easily complete during that downtime, you might want to lower your overhead and perform blocking LOAD operations.

Understanding the Version Log

The version log stores blocks of data that have changed due to INSERT, UPDATE, DELETE, and certain TMU operations, so you can store data while you completely load and verify a new revision of the database. You can make that revision available to the users after the entire load process is complete or at periodic intervals, as described in "Load with Periodic Commit" on page 6-5. You can also freeze the version available to users, as described in "Understanding Frozen Versions" on page 6-13. You can use the version log as a traditional write-ahead log by using the high recoverability option, as described in "Loading Data with the High Recoverability Option" on page 6-16.

The data in the version log is merged into the database files by the vacuum cleaner daemon, or by the TMU if you use the high recoverability option. The vacuum cleaner daemon or the TMU then frees the space in the version log for reuse. For more information on the vacuum cleaner, refer to "Vacuum Cleaner Daemon" on page 1-13.

Structure of the Version Log

The version log resides in a single segment that can contain up to 250 separate physical storage units (PSUs). Each PSU maps to an operating-system file. Physical storage of database files is divided into blocks. Each database block in Red Brick Decision Server uses 8 kilobytes of disk space. The following figure shows a database with five blocks of data.

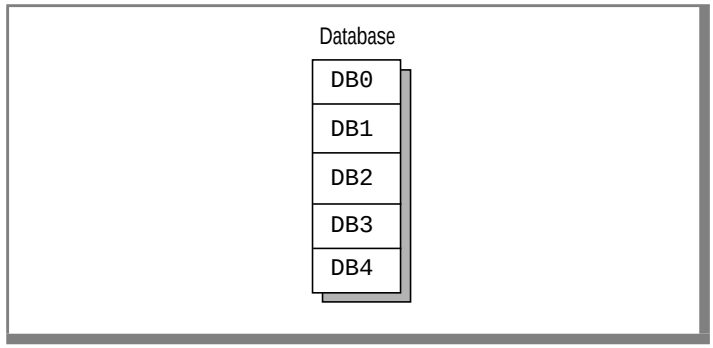


Figure 6-2
Database Blocks

The first block in the database is named DB0, and the last block is named DB4. If a query reads the whole database, it reads the blocks in the following order:

```
(DB0), (DB1), (DB2), (DB3), (DB4)
```

Suppose this database is versioned and therefore contains a version log. When a transaction changes block DB0 of the database and commits the change, the new version of that block is stored as revision number 1 of that block in the version log.

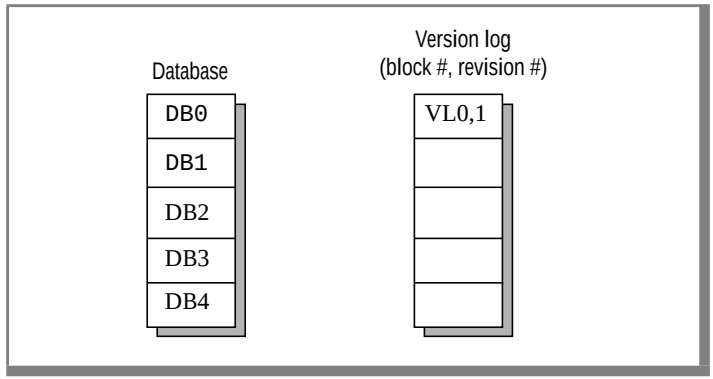


Figure 6-3
Version Log After First Transaction

Now if a query reads the whole database, it reads the following blocks:

```
(VL0,1), (DB1), (DB2), (DB3), (DB4)
```

If another transaction changes block 0 and block 2 of the database, the result is as follows.

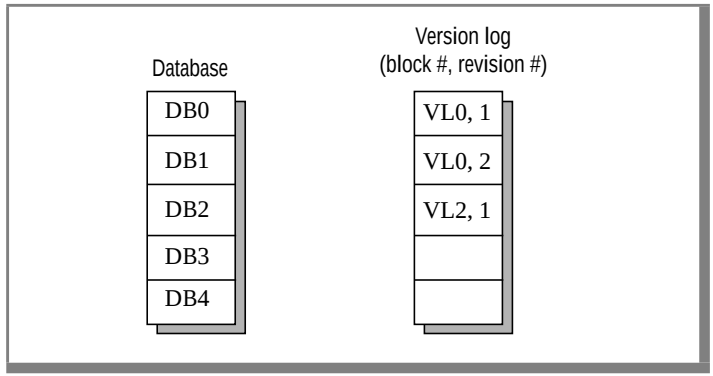


Figure 6-4 Version Log After Second Transaction

New transactions read the latest revision of the database. Because block 0 contains two revisions, the latest revision is revision number 2 in this example. Therefore if a query reads the whole database, it reads the following blocks:

```
(VL0,2), (DB1), (VL2, 1), (DB3), (DB4)
```

This explanation assumes the vacuum cleaner has not cleaned any blocks throughout the duration of this example. If it had, those blocks would have been moved from the version log to the database files. For more information on the vacuum cleaner, refer to "Controlling the Vacuum Cleaner" on page 6-25.

Versioned DELETE Operations

When performing DELETE operations on a versioned database, any deleted blocks are treated as changed blocks and are written to the version log. If you are deleting a small number of blocks relative to the number of blocks in a segment or a table, this arrangement is fine. This type of versioned DELETE operation allows users to query a table for the duration of the operation.

However, a versioned delete of a whole segment or table might perform slowly because it creates a new version of each block that is deleted in the version log, which must also subsequently be written to the database files. This situation increases both the necessary size of the version log and the I/O for the transaction. Furthermore, because the blocks in the table exist as empty blocks, a subsequent load operation into that table creates yet another version of each block in the version log.

Deleting everything in a table is better done as a nonversioned transaction or with an ALTER SEGMENT CLEAR operation, a blocking (nonversioning) operation that resets the segments to have no allocated blocks.

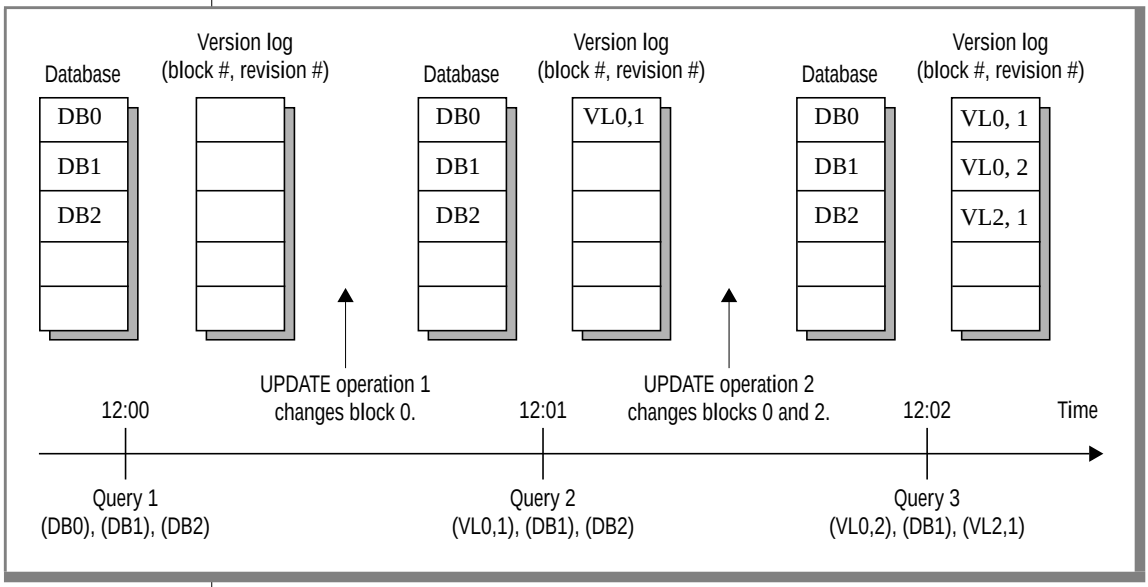
Understanding Frozen Versions

To ensure that all queries read the same revision of the database, use the frozen version mode. This mode allows you to set the current version of the database as the default read revision to be used by all subsequent queries. If the version is not frozen, a query on a versioned database reads the latest revision of the database at the time the query begins. Thus it is possible for even closely timed queries to access different data sets due to changes caused by updates in the interim, as the following example shows.

Example

Consider three queries that take place one minute apart. After the first query begins, an UPDATE operation commits its changes to the database. After the second query begins, another UPDATE operation commits its changes. Then the third query begins. Without the frozen version feature, each of these queries accesses a different revision of the database, as the following figure illustrates.

Figure 6-5
Query Behavior Without Frozen Versions



In this example, query 1 reads only the blocks in the database files, but query 2 and query 3 read some blocks from the database files and some from the version log. Each query reads the latest committed revision of the database, regardless of whether it currently resides in the database files, the version log, or a combination of the two.

If you freeze the version with an ALTER DATABASE FREEZE QUERY REVISION statement at or before 12:00, each subsequent query reads the same blocks, (DB0), (DB1), and (DB2). This arrangement ensures consistent results.

In frozen version mode, only the following modifications can be made to the database:

- Versioning operations with the SET USE LATEST VERSION ON statement
- Any operation on a temporary table because these tables are local to the session
- Any changes to a table that do not affect its structure: for example, adding a comment
- Operations that do not affect existing database objects, such as CREATE TABLE and CREATE SEGMENT

To use the following statements on a versioning database in frozen version mode, first override frozen versioning with the SET USE LATEST REVISION ON statement. For more information about turning frozen versions on or off, refer to "Controlling Frozen Versions" on page 6-21.

The following operations are allowed in frozen version mode:

- CREATE TABLE
- CREATE SEGMENT
- CREATE VIEW
- CREATE INDEX... DEFERRED
- CHECK TABLE
- CHECK INDEX
- DROP TABLE for temporary tables
- All DML commands
  - SELECT
  - UPDATE
  - INSERT
  - DELETE

The following operations are not allowed in frozen version mode:

- ALTER TABLE
- ALTER INDEX
- ALTER SEGMENT
- CREATE INDEX
- All DROP commands except those for temporary tables

Loading Data with the High Recoverability Option

If you need high recoverability during loading but do not need the ability to execute concurrent queries, use the SET TMU VERSIONING RECOVER statement to use the version log as a traditional write-ahead log.

When the loader is in high recoverability mode, it holds a blocking lock on the table that prevents concurrent queries. Therefore, loading with the high recoverability option is suitable for large nightly loads that have exclusive access to the table. This option is not suitable for environments that have continuous data feeds that allow concurrent queries.

As with other versioning options, loading with the high recoverability option writes the data to the version log first. Then the TMU (not the vacuum cleaner daemon) moves the changed blocks from the version log to the target database table when the load commits. If you use the Parallel TMU with the high recoverability option, it both loads and cleans in parallel.

The high recoverability option is not available for REORG or OFFLINE LOAD operations. Loading with the high recoverability option cannot be performed when the query revision is frozen.

Controlling Versioning

In general, to create and enable a versioned database, complete the following steps. Each of these steps is described in detail in the sections that follow. Information on increasing the size of the version log and using the frozen version mode appears in subsequent sections.

To create and enable a versioned database

1. Create the segment in which the version log will reside with a CREATE SEGMENT statement.
2. Ensure that no users are connected to the database. If necessary, you can perform an ALTER SYSTEM QUIESCE operation and ALTER SYSTEM CANCEL USER SESSION operations.

3. Create the version log with an ALTER DATABASE CREATE VERSION LOG statement.

   This statement allocates all of the physical space for the version log, so it might take some time to complete. All other users must be off the system before you can execute this statement, and all other connections to the database are refused while the version log is initializing

4. Start versioning with an ALTER DATABASE START VERSIONING statement.

This procedure sets up a versioned database. To run server operations as versioned transactions for concurrency, set versioning on by entering:

- OPTION VERSIONING ON in the **rbw.config** file to control all user sessions
- SET VERSIONING ON statement to control the current RISQL session

To run TMU operations as versioned transactions, set versioning on by entering:

- OPTION TMU_VERSIONING ON in the **rbw.config** file to control all user sessions
- SET TMU VERSIONING ON statement to control the current RISQL session

To set versioning on for high recoverability rather than concurrency, use the RECOVER keyword instead of ON.

If versioning is set to ON or RECOVER and no version log exists in the database or the database is not enabled for versioning (with ALTER DATABASE START VERSIONING), any transaction that attempts to modify the database fails with an error.

Creating the Version Log

Minor changes to data throughout the day do not require a large version log. As the size of the version log and the query demands on the modified data increase, configuration and sizing are more important.

The main performance cost of the version log is I/O because the data is first written to the version log and then written back to the database by the vacuum cleaner. To minimize the performance cost of the version log, configure it on high-performance devices.

For maximum performance, Informix recommends that you locate your version log in the following manner:

- On a dedicated storage subsystem separate from the rest of the database, ideally with several separate disk drives
- With disk striping or RAID level-0 or level-1 devices

  Placing the version log on several disks or on RAID devices allows I/O to occur from several disks at one time, thus having little or no performance impact on query operations.

Create the segment in which the version log will reside with a CREATE SEGMENT statement. To ensure that the disk space is available, all of it is allocated when the version log is created. The amount of data that will be written to the version log is directly related to the number of blocks in the existing database that are modified. New blocks are not written to the version log. They are written directly to the database files.

The size needed for the version-log segment is equal to the number of changed blocks in your database (at 8 kilobytes per block), plus about 20 percent for overhead, to equal the total kilobytes, or MAXSIZE. This number is difficult to estimate, however. Informix recommends that you set up a realistic test database and measure it directly. For information on determining the size of your version log, refer to "Monitoring the Version Log" on page 6-23.

Little performance cost accrues from overestimating the size of the version-log segment. However, if you underestimate and it runs out of space during a transaction, the transaction aborts. You lose the changes made by the aborting transaction although the aborting transaction does not corrupt the database.

The vacuum cleaner daemon cleans a revision out of the version log when no queries are accessing any previous revisions. Long-running queries prevent cleaning and can increase the size of the version log. Frozen version mode functions essentially as a long-running query and likewise increases the size of the version log.

Cleaning the Version Log

To clean the version log to prepare for a backup or similar operation, use the ALTER DATABASE CLEAN VERSION LOG statement. This statement is a blocking statement. Set a read lock on the database before executing it. For the syntax of this statement, see the *SQL Reference Guide.*

You can also use the ALTER DATABASE CLEAN VERSION LOG statement to clean the version log before dropping it. If you drop the version log, you do not need to lock the database. Stop versioning, clean the version log, and then drop it.

To clean the version log before restoring the system from a backup, use the REMOVE DAMAGED SEGMENTS option of the ALTER DATABASE CLEAN VERSION LOG statement. This option removes damaged segments from the version log, which can cause information loss. If this option is not set, damaged segments remain in the version log, which prevents you from dropping the version log.

The ALTER DATABASE CLEAN VERSION LOG statement returns when the version log is empty or when the statement encounters a damaged segment in the version log. Always verify that the version log is empty before proceeding with a backup operation.

To verify that the version log is empty:

```
select dbname FROM dst_databases
where dbname = 'Database_name'
and current_revision <> latest_merged_revision;
```

Database_name is the name of your database. When this query returns no rows, the version log is empty.

If the version log does not empty, queries might be accessing older revisions and preventing cleaning. Check the **DST_COMMANDS** table for the queries with the minimum READ_REVISION values. If appropriate, you can then terminate the queries from the ADMIN database.

If the query revision is frozen, the vacuum cleaner cannot finish cleaning the version log until the database is unfrozen.

For instructions on backing up a database, see "Backup and Recovery" on page 6-24.

Dropping the Version Log and Adding Space

To add space to the version log for a database, first drop it, and then create it again, specifying a larger size:

1. Stop versioning on the database by entering the following statement:

   ```
   alter database stop versioning;
   ```

   This statement disallows any new versioning transactions.

2. Clean the version log:

   ```
   alter database clean version log;
   ```

   For more information, see "Cleaning the Version Log" on page 6-19.

3. Wait until the vacuum cleaner finishes emptying the version log. The version log is empty when the value of the **CURRENT_REVISION** column in the **DST_DATABASES** table is equal to the value of the **LATEST_MERGED_REVISION** column, as in the following example:

   ```
   RISQL> select current_revision, latest_merged_revision
      from dst_databases;

   CURRENT_REV LATEST_MERG
             1           1
   ```

4. Ensure that no users are connected to the database. If necessary, you can perform an ALTER SYSTEM QUIESCE operation, an ALTER SYSTEM CANCEL USER SESSION operation, or both.

5. Drop the version log.

   ```
   alter database drop version log;
   ```

6. Add more space to the segment with ALTER SEGMENT operations, by adding PSUs to the version log segment or increasing the maximum size of the existing PSUs.

7. Create the version log again, as in the following example:

```
alter database create version log in
   version_log_segment;
```

8. Start versioning on the database again.

```
alter database start versioning;
```

Controlling Frozen Versions

Freezing the version causes the database server to read the current revision number and sets that number as the default read-revision number to be used by all subsequent queries unless specifically overridden in a session. To freeze the version, you must have the database administrator role, and you must first enable versioning.



Warning: *Freezing the current version prevents all subsequent revisions from being cleaned by the vacuum cleaner and increases storage space requirements of the version log.*

Freezing the Database

Freeze the database at the current revision with the following statement:

```
alter database freeze query revision;
```

If the query revision is already frozen, the statement fails. The frozen version mode is persistent and survives system failure and restart.

After this statement executes, the database server displays the revision number chosen to be the query revision. You can also view the read revision number at any time in the **QUERY REVISION** column of the **DST_DATABASES** table. If the revision is not frozen, the value of this column is NULL.

To find the revision number of the database that your RISQL command is accessing:

```
select read_revision from dst_commands;
where command like 'select read_revision from dst_commands%';
```

Overriding Frozen Versions

To override the frozen version mode and use the current revision for a specific user session, use the following SET command:

```
set use latest revision on;
```

To update a database while in frozen query mode, you must be using a session that has this SET command ON.

Unfreezing the Database

To unfreeze the database for all user sessions, enter the following statement:

```
alter database unfreeze query revision;
```

Unloading Data with Frozen Versions

The TMU unload operation essentially functions as a query. By default, the TMU uses the latest revision when unloading tables rather than the current query revision (frozen version). To specify the frozen version include the USING QUERY REVISION option in the TMU UNLOAD command. If there is no frozen version, the UNLOAD command uses the latest revision and does not return a message.

Using Frozen Versions and High Recoverability

You cannot use frozen versions and the load with high recoverability option at the same time on the same table. Loading with high recoverability is an exclusive operation. Using both features at the same time on the same table can result in an error.

Maintaining a Versioned Database

After you have a version log set up with all of the space allocated, little maintenance is required. However, maintenance of a versioned database involves several special considerations.

Monitoring the Version Log

Monitor the activity of the version log to determine whether you have over- or under-allocated disk space. If the version log is filling up, consider allocating more space or reconfiguring it. For information on adding space, refer to "Dropping the Version Log and Adding Space" on page 6-20. For information on configuration, refer to "Creating the Version Log" on page 6-18.

To monitor the space being used by the version log, refer to the following columns of the **DST_DATABASES** table.

Column Name	Description
VERSION_LOG_USED	The amount of disk space (in kilobytes) used in the version log for new versions of database blocks. NULL if version log does not exist.
VERSION_LOG_AVAILABLE	The amount of free disk space (in kilobytes) available in the version log. NULL if version log does not exist.
VERSION_LOG_MAXIMUM_USED	The maximum amount of disk space (in kilobytes) used in the version log. Also known as the *high water mark*. Can be reset with the ALTER SYSTEM RESET STATISTICS statement. NULL if version log does not exist.

For more information, refer to "Monitoring Database Activity with Dynamic Statistic Tables" on page 8-6.

Event log messages also provide monitoring information. When the version log reaches a capacity of 90 percent, warning messages are sent to the event log. For more information on event logs, refer to Chapter 8, "Managing Database Activity in an Enterprise."

Backup and Recovery

Before you perform a backup or restore operation, or when you copy or move a versioned database, the version log should be empty. With versioning for recoverability, perform a backup after loading the data. With versioning for concurrency, perform a backup after completing the following procedure.

To prepare for backup when using versioning for concurrency

1. Generate the list of PSUs to back up.

   ```
   select physical_location from rbw_storage;
   ```

   This query returns the list of PSUs, their sizes, and their last update times.

2. Lock the database using a read lock.

   ```
   lock database read;
   ```

   The read lock allows concurrent queries but not updates. For more information, see "Manual Table or Database Locks" on page 9-6.

3. Clean the version log.

   ```
   alter database clean version log;
   ```

   For more information, see "Cleaning the Version Log" on page 6-19.

4. Verify that the version log is empty.

   ```
   select dbname from dst_databases
   where dbname = 'Database_name'
   and current_revision <> latest_merged_revision;
   ```

   Database_name is the name of your database. When this query returns no rows, you can safely back up the database.

If the database is corrupted such that the version log does not empty itself, call Informix Customer Support. For instructions on dropping the version log, refer to "Dropping the Version Log and Adding Space" on page 6-20.

Controlling the Vacuum Cleaner

The vacuum cleaner process moves the data committed in the version log to the database files. There is one vacuum cleaner for each versioned database. After data is committed to the version log, the vacuum cleaner waits for any queries that are accessing previous revisions of the data to complete and then writes the newly modified data back to the database files. Then it frees the space in the version log for use by new transactions.

The vacuum cleaner performs its work in the background, cleaning out the version log without adversely affecting anything else on the database server. If vacuuming seems to degrade performance, you can use manual controls to start and stop the vacuum cleaner. Under normal conditions, you should not stop the vacuum cleaner because it might cause the version log to run out of space.

If you perform a blocking transaction that modifies a versioned database (for example, a LOAD operation with SET TMU VERSIONING OFF), the vacuum cleaner cleans the version log before the transaction executes. The blocking transaction then performs its work solely on the database files, not on the version log.

Because the vacuum cleaner waits for queries to finish reading previous versions before writing the new versions back to the database files, you might notice that the vacuum cleaner is active on your system even when no one is connected to the database.

To manually disable the vacuum cleaner, enter the following statement:

```
alter database stop cleaning;
```

This action allows you to control when the cleaning occurs. Because the process is I/O intensive, it can potentially cause a performance bottleneck if queries also need to perform extensive of I/O on the same devices.



Warning: *If you manually stop the vacuum cleaner, the version log might run out of disk space. If this situation occurs, the transaction is aborted, changes to the version log are discarded, and the database remains in its previous state.*

If you stopped the vacuum cleaner manually and want to start it back up when the system is quiet, use the following statement:

```
alter database start cleaning;
```

Example: Creating a Versioned Aroma Database

The **Aroma** sample database is installed when you install Red Brick Decision Server. **Aroma** is a retail database with approximately 69,000 rows of data. The following procedure enables versioning for concurrency in the **Aroma** database:

1. Create the **Aroma** database. If you have deleted **Aroma**, re-create it as described in Appendix A, "Example: Building a Database."
2. Create the segment in which the version log will reside. From the RISQL prompt, enter the following statement:

   ```
   RISQL> create segment versionlog
      storage 'version.log' maxsize 10000;
   ```

3. Create the version log in the new segment by entering the following statement:

   ```
   RISQL> alter database create version log in versionlog;
   ```

4. Start versioning by entering the following statement:

   ```
   RISQL> alter database start versioning;
   ```

5. Enable versioned transactions globally with the OPTION VERSIONING ON configuration file parameter or for a session with the following statement:

   ```
   RISQL> set versioning on;
   ```

You can now query and update the database concurrently.

Chapter

7

Providing Database Access and Security

Adding Database Users 7-3
Creating Operating-System Accounts for Users. 7-4
Granting Database Access 7-5
Changing Passwords 7-6

Granting Access with System Roles 7-7
DBA, RESOURCE, and CONNECT Capabilities 7-7
Granting and Revoking the DBA and RESOURCE System Roles . . 7-8

Granting Database Object Privileges 7-9

Granting Access with Role-Based Security 7-10
Task Authorizations 7-12
Role Capabilities 7-14
Creating Roles 7-15
Granting Task Authorizations 7-16
Granting Object Privileges to Roles 7-16
Granting Roles 7-18
Revoking Task Authorizations, Object Privileges, and Roles . . . 7-21
Tracking Role Authorizations and Members 7-23

Administering Password Security. 7-26
Enforcing Password Changes 7-28
Warning Users of Password Expiration 7-30
Limiting Reuse of Previous Passwords. 7-31
Limiting Frequency of Password Changes 7-32
Enforcing Password Complexity and Length 7-33
Locking User Accounts After Failed Connection Attempts 7-36
Locked Account Status 7-37

In This Chapter

After creating a database, the database administrator has access through a single user account. In order to provide access to other users, the database administrator needs to give them the ability to connect to the database and to perform tasks within the database. To maintain security, the administrator must decide which users can connect to the database and what database actions they can perform. The administrator then implements the appropriate access scheme by creating database names and passwords and assigning each user only the relevant capabilities.

This chapter discusses database access and security and includes the following sections:

- Adding Database Users
- Granting Access with System Roles
- Granting Database Object Privileges
- Granting Access with Role-Based Security
- Administering Password Security

Adding Database Users

If the users at your site access the database via network-connected client applications, you need not create operating-system accounts for those users. However, if they access the database locally, via the RISQL Entry Tool or RISQL Reporter running on the same computer, you need to create operating-system accounts that provide individual login access to the system.

Before any user can access the database—via client tools, the RISQL Entry Tool, or the RISQL Reporter—you must grant that user database access with the SQL GRANT statement.

Creating Operating-System Accounts for Users

UNIX

After Red Brick Decision Server has been installed and you are ready to add users to the database, make sure that:

- The **redbrick_dir/bin** directory is in the path for each user.
- The **RB_CONFIG** and **RB_HOST** environment variables are initialized correctly and are accessible by each user.
- The **RB_PATH** environment variable is initialized correctly and is accessible by each user (in a single-warehouse database environment) or each warehouse database has a logical name definition in the **rbw.config** file.

The following lines illustrate path and environment variable settings in a profile file for a Korn shell user account:

```
...
PATH=$PATH:/usr/redbrick_dir/bin;export PATH
RB_CONFIG=/usr/redbrick_dir; export RB_CONFIG
RB_HOST=RB_HOST; export RB_HOST
...
```

♦

WIN NT/2000

After Red Brick Decision Server has been installed and you are ready to add users to the database, make sure that:

- The **redbrick_dir\bin** directory is in the path for each user. This should always be the case because the installation puts the **redbrick_dir\bin** directory in the system path.
- The following environment variables are set in the Windows Registry for the Red Brick Decision Server service, which is in ***HKEY_LOCAL_MACHINE**SOFTWARE*\RedBrick\RedBrickWarehouse*DefaultServer***:
  - **RB_CONFIG**
  - **RB_EXE**
  - **RB_HOME**
  - **RB_PATH**
- The **RB_HOST** environment variable is set either in the user environment or in the Windows Registry. ♦

For more information about the environment variables, refer to Chapter 2, "Key Concepts."

No special permissions or privileges are required for RISQL Entry Tool or RISQL Reporter user accounts. For more information about database access from these tools, refer to the *RISQL Entry Tool and RISQL Reporter User's Guide*.

Granting Database Access

Whether users connect to the database via client tools, the RISQL Entry Tool, or the RISQL Reporter, you must give each user database access by using the SQL GRANT statement.



Tip: *Some ODBC client applications, including Microsoft Access and Visual Basic, require that you grant resource privileges with the SQL GRANT statement for each ODBC-application user on your system.*

To add a new user to the database

1. Verify that the user has either a system account or access through a client tool.
2. As the database administrator (or as member of the DBA system role or a user with USER_MANAGEMENT task authorization), start a RISQL session.
3. Use the SQL GRANT CONNECT statement to add the user name to the database and assign a password. The user is authorized to connect to the database, change (own) password, use PUBLIC macros, and access PUBLIC tables. These tasks make up the CONNECT system role.
4. Decide which database capabilities you want the user to have and then use the GRANT statement to make the user a member of the appropriate system role, to assign the appropriate object privileges, and to grant privileges indirectly through assignment to a role.

   For descriptions of these tasks see "Granting Access with System Roles" on page 7-7, "Granting Database Object Privileges" on page 7-9, and "Granting Access with Role-Based Security" on page 7-10.

To drop database users from a database, use the REVOKE CONNECT statement.

For information about the GRANT, REVOKE, and CREATE ROLE statements, refer to the *SQL Reference Guide*.

Examples

The following GRANT CONNECT statement creates the database user name **drew** and assigns the password **instructor**:

```
grant connect to drew with instructor ;
```

Now **drew** can connect to the database, change her own password, use PUBLIC macros, and access PUBLIC tables. To perform any other operations, the user **drew** must be granted the RESOURCE or DBA system role or object privileges.

The following REVOKE CONNECT statement removes **drew** from the database:

```
revoke connect from drew;
```

Changing Passwords

With the GRANT CONNECT statement, database users can change their own passwords, and the database administrator can change the password for any user. To change a password, simply specify the database user name and supply a new password.

A password can be any valid database identifier or string literal, as defined in the *SQL Reference Guide*. Optional restrictions can be placed on passwords by setting the password parameters in the configuration file. For more information, refer to "Administering Password Security" on page 7-26.

Example

The following GRANT CONNECT statement changes the password for the user **drew** to **se2cure**. Either **drew** or the database administrator can issue this statement.

```
grant connect to drew with se2cure ;
```

Granting Access with System Roles

All Red Brick Decision Server databases have three predefined system roles:

- The CONNECT system role allows users to connect to the database, change their own passwords, use PUBLIC macros, and access PUBLIC tables. Users become members of the CONNECT system role when they are added to the database.
- The RESOURCE system role includes the capabilities of the CONNECT system role. It also allows users to create database objects and to modify, drop, and grant access to those objects.
- The DBA system role provides the capabilities of the CONNECT and RESOURCE system roles. It also allows users to access and modify all objects in the database and to affect the structure and security of the database.

DBA, RESOURCE, and CONNECT Capabilities

The following table defines the actions permitted to members of the DBA, RESOURCE, and CONNECT system roles, independent of any object privileges.

	System Roles		
Tasks Permitted	**DBA**	**Resource**	**Connect**
GRANT/REVOKE system roles	Yes	No	No
CREATE database objects	Yes	Yes	No
ALTER database objects	Yes	Yes if object creator	No
DROP database objects	Yes	Yes if object creator	No
SELECT data	Yes	Yes if object creator	No
Modify data	Yes	Yes if object creator	No
GRANT/REVOKE object privileges	Yes	Yes if object creator	No

(1 of 2)

	System Roles		
Tasks Permitted	**DBA**	**Resource**	**Connect**
LOCK the database	Yes	No	No
UPGRADE the database	Yes	No	No
EXPORT data	Yes	No	No
REORG a table	Yes	Yes if table creator	No
Perform offline loads	Yes	Yes if table creator and using own segments	No

(2 of 2)

For information about using role-based security to break down the tasks of the system roles and recombine them in new roles, refer to "Granting Access with Role-Based Security" on page 7-10.

Granting and Revoking the DBA and RESOURCE System Roles

As a member of the DBA system role, you can use the GRANT statement to grant the DBA and RESOURCE system roles to other database users. Granting users the RESOURCE or DBA system role allows them to perform the tasks assigned to those roles within the database.

As a member of the DBA system role, you can revoke the DBA and RESOURCE system roles from a user at any time with the REVOKE authorization and role statement. For more information about the GRANT and REVOKE statements, refer to the *SQL Reference Guide*.

Examples

The following GRANT statement grants the RESOURCE system role to **drew**, who has already been granted CONNECT. The database user **drew** will be able to create database objects and access and modify these objects.

```
grant resource to drew ;
```

The following REVOKE statement removes the user **bob** from the DBA system role:

```
revoke dba from bob ;
```

Granting Database Object Privileges

An object privilege allows a user to select or modify data from a specific database object, such as a table. The five object privileges are as follows:

- SELECT
- INSERT
- UPDATE
- DELETE
- ALL PRIVILEGES

As a member of the DBA system role (or as a RESOURCE member and creator of the object), you can use the GRANT statement to grant object privileges to database users. Object privileges are granted to one or more specified users or to all users, specified as PUBLIC. Users must be granted the CONNECT system role and assigned a password before being granted object privileges. Object privileges can be removed at any time with the REVOKE statement.

The following table defines the actions permitted on database objects for a user who is a member of the DBA system role, created the object, and has been granted object privileges, and for all others.

	Users			
Object Privileges Permitted	**DBA**	**Creator [1]**	**Grantee**	**PUBLIC**
SELECT from object	Yes	Yes	Yes	No
INSERT into object	Yes	Yes	Yes [2]	No

[1] Members of the RESOURCE system role have all object privileges on tables they create.

[2] To insert rows, users must have INSERT and SELECT privilege on the object.

(1 of 2)

Object Privileges Permitted	Users			
	DBA	Creator [1]	Grantee	PUBLIC
UPDATE object	Yes	Yes	Yes	No
DELETE from object	Yes	Yes	Yes	No
Use PUBLIC macro	Yes	Yes	Not applicable	Yes

[1] Members of the RESOURCE system role have all object privileges on tables they create.

[2] To insert rows, users must have INSERT and SELECT privilege on the object.

(2 of 2)

For a complete discussion of the GRANT and REVOKE statements, refer to the *SQL Reference Guide*.

Example

This example illustrates how the user **curly** (with RESOURCE) can grant the SELECT privilege on a table named t1 that he created to the user **moe** (who has already been granted CONNECT).

```
grant select on t1 to moe ;
```

Granting Access with Role-Based Security

Role-based security provides more control and flexibility in managing users and their capabilities than do the predefined system roles. With role-based security, you not only have the predefined RESOURCE and DBA system roles, but you also can grant separate tasks, recombine tasks in new roles, and group database users in custom, or user-created, roles.

A user-created role can consist of any combination of the following items:

- Task authorizations, as defined in the table on page 7-12
- Object privileges, as defined in "Granting Database Object Privileges" on page 7-9
- Database users
- Other roles

After creating a role, you can grant it to additional users who are not already members of that role. The grantee becomes a member of the role and has all of its authorizations and privileges. You can alter a role at any time by granting or revoking task authorizations, object privileges, users, and other roles.

If a user is granted membership in a role, that user is a direct member of the role. If a role (**role1**) is granted to another role (**role2**), the second role (**role2**) is an indirect member of the granted role (**role1**).

In general, use the RESOURCE and DBA system roles and object privileges whenever appropriate, and create and use custom roles only when your database administration tasks and security would benefit from the added flexibility.

This section discusses role-based security in terms of:

- A list and description of the task authorizations
- A description of role capabilities, with examples to illustrate role flexibility
- Creating roles
- Granting task authorizations to users and roles
- Granting roles
- Granting object privileges to roles
- Revoking task authorizations, object privileges, and roles
- Tracking role authorizations and members

Task Authorizations

The following table lists the task authorizations included in the DBA system role.

Task Authorization	Definition
ACCESS_ADVISOR_INFO	Query the Advisor system tables. For information about the Advisor, refer to the *Informix Vista User's Guide*.
ACCESS_ANY	Select data from all database objects and access private user information (such as private macros) in the system tables.
ACCESS_SYSINFO	Query the dynamic statistic tables for statistics about database activity. For information about the dynamic statistic tables, refer to "Monitoring Database Activity with Dynamic Statistic Tables" on page 8-6.
ALTER_ANY	Alter columns, indexes, macros, segments, synonyms, tables, and views.
ALTER_SYSTEM	Issue the ALTER SYSTEM statement to perform database administration tasks.
CREATE_ANY	Create any object, including those that use the resources of another user. For example, create an index on the table of another user or create a table that resides in the segment of another user.
DROP_ANY	Drop objects created by any user.
EXPORT	Grant authority to export the results of an arbitrary query to a user-specified file with the EXPORT statement. This capability is intended primarily for DBAs and system administrators, not the general user community. This authorization allows the user to create files on the database host system as the user under which the database server is configured (usually **redbrick**).
GRANT_TABLE	Grant object privileges to database users and roles.
IGNORE_QUIESCE	Grant object access to a quiesced database. Makes it possible to load data or perform other administrative activities while the database is still in a quiesced state.
LOCK_DATABASE	Lock the database.
MODIFY_ANY	Insert, update, delete, and load any data.

(1 of 2)

Task Authorization	Definition
OFFLINE_LOAD	Use any segment as a working segment for offline loads. Synchronize segments after offline loads.
PUBLIC_MACROS	Create and drop PUBLIC macros.
REORG_ANY	Reorganize any table or index.
ROLE_MANAGEMENT	Create, drop, grant, revoke, and alter roles.
UPGRADE_DATABASE	Upgrade the database.
USER_MANAGEMENT	Create database users and change passwords with GRANT CONNECT. Drop database users with REVOKE CONNECT. Specify the default priority of a user's sessions with ALTER USER or GRANT CONNECT.

(2 of 2)

The following table defines the task authorizations included in the RESOURCE system role.

Task Authorization	Definition
ALTER_OWN	Alter own indexes, segments, and tables.
ALTER_TABLE_INTO_ANY	Alter own tables into other users' segments.
CREATE_OWN	Create own objects (indexes, private macros, segments, synonyms, tables, and views).
DROP_OWN	Drop own objects.
GRANT_OWN	Grant object privileges on own objects to other users.
TEMP_RESOURCE	Create temporary tables.



Tip: *You cannot grant task authorizations or object privileges to the DBA and RESOURCE system roles because system roles cannot be altered. However, you can grant a system role to a user-created role.*

For the complete syntax of the GRANT authorization and role statement and the ALTER SYSTEM statement, refer to the *SQL Reference Guide*.

Role Capabilities

In addition to the GRANT and REVOKE capabilities available with the predefined system roles, you can also use role-based security to:

- Grant a task authorization to one or more database users. For example:

```
grant lock_database to db_user ;
```

- Create a user-defined role. For example:

```
Create role table_select_role ;
```

- Grant an object privilege to a user-created role. Any member of the role then has the privilege. For example:

```
grant select on period to table_select_role ;
```

- Grant a task authorization to a user-created role. Any member of the role can then perform the task. For example:

```
grant upgrade_database to db_management_role ;
```

- Grant a user-created role to one or more database users. The users then become direct members of the role. The user-created role can consist of any combination of users, task authorizations, object privileges, or other roles, or it can be empty. For example:

```
grant table_select_role to db_user1, db_user2,
   db_user3 ;
```

- Grant a user-created role to another user-created role. Each user-created role can consist of any combination of users, task authorizations, object privileges, or other roles. For example:

```
grant table_select_role to marketing_role ;
```

  Members of the **marketing_role** become indirect members of the **table_select_role** and have all its capabilities.

- Grant a system role to a user-created role. For example:

```
grant resource to marketing ;
```

  All members of the **marketing** role become indirect members of the RESOURCE system role and can perform the tasks of this role.

- Revoke an object privilege from a user-created role. For example:

```
revoke select on period from table_select_role ;
```

- Revoke a task authorization from a user-created role or a database user. For example:

```
revoke upgrade_database from db_management_role ;
```

- Drop a user-created role. For example:

```
drop role table_select_role ;
```

Creating Roles

Use the CREATE ROLE statement to create a role and optionally grant the role to users and other roles. Listing users in a CREATE ROLE statement makes the users direct members of the role. Each database user can be a direct member of up to 16 roles.

After creating a role, use the GRANT authorization and role statement to grant task authorizations and other roles to the new role and to grant the new role to database users and other roles.

Use the GRANT privilege statement to grant object privileges to a role.

For complete syntax of the CREATE ROLE and GRANT statements, refer to the *SQL Reference Guide*.

Example

The following CREATE ROLE statement creates the **security_management** role and assigns **chris** and **judy** as direct members:

```
create role security_management for chris, judy ;
```

Granting Task Authorizations

Use the GRANT authorization and role statement to grant task authorizations to database users and to user-created roles. Granting a task authorization to a user allows the user to perform the task. Granting a task authorization to a role allows all direct and indirect members of the role to perform the task.

Examples

The following GRANT statement allows users **maria**, **john**, and **joe** to upgrade and lock the database:

```
grant upgrade_database, lock_database to maria, john, joe ;
```

The following example illustrates how to create a role for a group of users and then grant task authorizations to the role:

1. Create the **security_management** role with users **chris** and **judy** as direct members.

   ```
   create role security_management for chris, judy ;
   ```

2. Grant the USER_MANAGEMENT, GRANT_TABLE, and ROLE_MANAGEMENT task authorizations to the **security_management** role.

   ```
   grant user_management, grant_table, role_management
   to security_management ;
   ```

Users **chris** and **judy** are direct members of the **security_management** role and can manage database users, grant object privileges, and manage roles. If necessary, the **security_management** role can later be granted to additional users and to other roles.

Granting Object Privileges to Roles

You can use the GRANT statement to grant an object privilege to a user-created role. Granting an object privilege to a role provides all direct and indirect members of the role with the privilege. For information about granting object privileges to users and for a list of the object privileges, refer to "Granting Database Object Privileges" on page 7-9.

For complete syntax of the GRANT privilege statement, refer to the *SQL Reference Guide*.

Example

The following example illustrates how to create the roles **table_select** and **marketing** and then grant the SELECT object privilege on four tables to the role **table_select**.

If an empty role is granted to database users, the role consists only of the users but has no specific tasks or privileges associated with it. Creating a role with only a list of users is useful for grouping users so that you can assign task authorizations or object privileges, either individually or as a role, to the entire group of users all at once. For example, suppose you want to assign select privileges on several database tables to the members of the marketing department. You can use the following steps:

1. Create two roles. For example, **table_select** and **marketing**:

```
create role table_select ;
create role marketing ;
```

2. Grant object privileges to the **table_select** role.

```
grant select on period to table_select ;
grant select on product to table_select ;
grant select on market to table_select ;
grant select on sales to table_select ;
```

3. Grant the **marketing** role to a group of users.

```
grant marketing to db_user1, db_user2, db_user3,
   db_user4 ;
```

4. Grant the **table_select** role to the **marketing** role.

```
grant table_select to marketing ;
```

Members of the **marketing** role become indirect members of the **table_select** role and can access the **Period**, **Product**, and **Sales** tables.

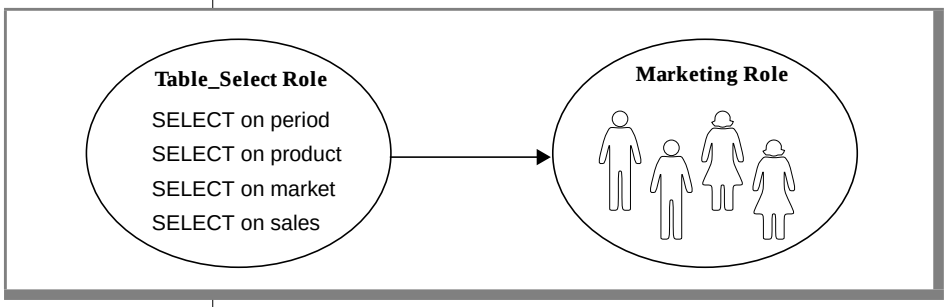


Figure 7-1 Indirect Granting of Table_Select Role

You can add new employees to the **marketing** role, giving them all the capabilities of that role with a single GRANT statement.

Granting Roles

Use the GRANT statement to grant roles to database users and to user-created roles. Granting a role to a user makes the user a direct member of the role. The user can perform all task authorizations and object privileges that have been granted to the role. Each database user can be a direct member of up to 16 roles.

When a role is granted to a user-created role, all members of the role receiving the grant become indirect members of the granted role and obtain all of its capabilities. Each database user can be an indirect member of an unlimited number of roles.

You cannot:

- Grant a role to a system role

  However, you can grant a system role to a user-created role
- Grant a role to itself
- Create a role indirection cycle

  For example, if you grant **Role1** to **Role2**, you cannot grant **Role2** to **Role1**.

Exercise caution when granting roles to users and other roles. You should always know the privileges of each user. By granting roles to other roles, you must keep track not only of the direct members but also of the indirect members. You can use the system tables to monitor role membership, task authorizations, and object privileges, as described in "Tracking Role Authorizations and Members" on page 7-23.

For complete syntax of the GRANT authorization and role statement, refer to the *SQL Reference Guide*.

Example

The following example illustrates how to create a role, grant task authorizations to the role, and then grant the role to a user. Suppose you need to allow a database user to upgrade a database whenever necessary. You can create a role specifically for this purpose, grant that role only the necessary task authorizations, and then grant the user membership in the role:

1. Create a role named **database_management**.

```
create role database_management ;
```

2. Grant the necessary subset of the DBA task authorizations, in this example LOCK_DATABASE and UPGRADE_DATABASE, to the **database_management** role.

```
grant lock_database upgrade_database to
database_management ;
```

3. Grant this new role to the user who will have this responsibility.

```
grant database_management to db_user ;
```

The user **db_user** becomes a member of the **database_management** role and can lock and upgrade the database.

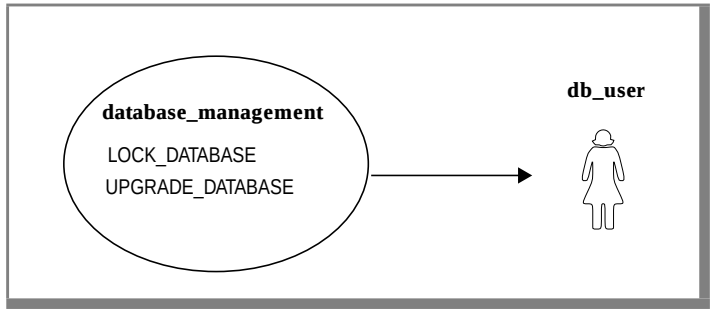


Figure 7-2 Indirect Membership in Role

With a single GRANT statement, you can later provide additional users with all the capabilities of the **database_management** role when the tasks assigned to it become too much for one person to handle.

Example

The following example illustrates how to create the **marketing** role to group users in the marketing department and the **object_management** role to group object management tasks, and then grant the **object_management** role to the **marketing** role. All members of the **marketing** role become indirect members of the **object_management** role.

1. Create the role **marketing** and make users **sudhir**, **nasi**, and **cody** direct role members.

```
create role marketing for sudhir, nasi, cody ;
```

2. Create the role **object_management**.

```
create role object_management ;
```

3. Grant task authorizations to the **object_management** role.

```
grant alter_any, public_macros, access_any,
   modify_any,drop_any, create_any to
   object_management ;
```

4. Grant the **object_management** role to the **marketing** role, which makes users **sudhir**, **nasi**, and **cody** indirect members of the **object_management** role and allows them to perform all tasks granted to this role.

```
grant object_management to marketing ;
```

Example

This example illustrates the concept of indirect role membership. Suppose you have three roles in your database with the following direct members and task authorizations.

As members of **Role2**, users Brian and Hedy can upgrade databases and reorganize tables. If you grant **Role1** to **Role2**, Brian and Hedy become indirect members of **Role1** and can now create and modify any objects. The reverse is not true. Kirsten and Susan cannot upgrade databases or reorganize tables.

As members of **Role3**, Emily and Elena can access and alter any database objects. If you grant **Role2** to **Role3**, Emily and Elena become indirect members of **Role2**. As members of **Role2**, they are also indirect members of **Role1**. Users Emily and Elena can now create and modify any database objects and can upgrade and reorganize databases, in addition to performing their **Role3** tasks.

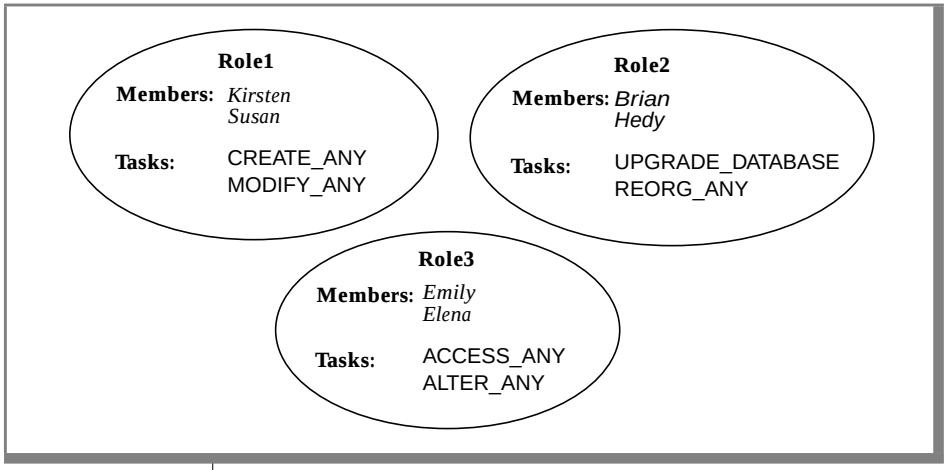


Figure 7-3 Role Members and Task Authorizations

If you do not want to allow user Emily to create objects, you cannot simply revoke the CREATE_ANY task authorization from Emily because Emily has this task authorization only through membership in **Role1**. Instead, you have the following choices:

- Revoke **Role3** from Emily, which would prevent her from performing all tasks.
- Revoke **Role2** from **Role3**, which would remove the link to **Role1**.
- Revoke **Role1** from **Role2**, which would disallow direct and indirect members of **Role2** from performing **Role1** tasks.
- Revoke the CREATE_ANY task authorization from **Role1**.

Revoking Task Authorizations, Object Privileges, and Roles

Use the REVOKE authorization and role statement to revoke task authorizations and roles. Use the REVOKE privilege statement to revoke object privileges. To remove a capability from a user, you must revoke each occurrence of the task or privilege from that user. For example, if a user has been granted both a task authorization and a role that contains that task authorization, you must revoke both the task authorization and either the role from the user or the task authorization from the role.

You cannot revoke task authorizations from system roles. System roles cannot be altered.

For the complete syntax of the REVOKE authorization and role and REVOKE privilege statements, refer to the *SQL Reference Guide*.

Example

This example shows how a user might be granted a task authorization multiple times and how to completely revoke the authorization from that user. Assume the user Ken is first granted the UPGRADE_DATABASE task authorization directly. Then Ken is granted the **database_management** role, which has also been granted the UPGRADE_DATABASE task authorization.

If you do not want Ken to upgrade databases but are unsure if he has the authorization to do so, check the RBW_USERAUTH table.

```
select grantee, grantor, upgrade_database
from rbw_userauth ;

GRANTEE                 GRANTOR      UPGR
DATABASE_MANAGEMENT     SYSTEM       Y
KEN                     DBA          Y
```

Revoke the UPGRADE_DATABASE task authorization from **ken**.

```
revoke upgrade_database from ken ;
```

To see if user **ken** can still upgrade databases, check the **RBW_USERAUTH** table again.

```
select grantee, grantor, upgrade_database
   from rbw_userauth ;

GRANTEE                  GRANTOR     UPGR
DATABASE_MANAGEMENT      SYSTEM      Y
KEN                      SYSTEM      R
```

The R in the preceding results indicates that **ken** has the authorization through a role. Revoke the **database_management** role from **ken**.

```
revoke database_management from ken ;
```

Query the **RBW_USERAUTH** table again to verify that **ken** can no longer upgrade databases.

```
select grantee, grantor, upgrade_database
   from rbw_userauth ;

GRANTEE                  GRANTOR     UPGR
DATABASE_MANAGEMENT      SYSTEM      Y
```

If **ken** belongs to multiple roles or if multiple roles have this task authorization, you have to determine which role grants user **ken** the task authorization and either eliminate his membership in that role or remove the authorization from it.

Instead of revoking the **database_management** role from **ken**, you can revoke UPGRADE_DATABASE from the **database_management** role. Either method prevents **ken** from performing the task. However, revoking a task from a role prevents *all* members of the role from performing the task.



Tip: *To drop a role from the database, use the DROP ROLE statement. If you drop a role, role members might still have indirect task authorization for some of the tasks of the dropped role. For more information about dropping a role, refer to* *"Roles" on page 9-66**. For complete syntax of the DROP ROLE statement, refer to the "SQL Reference Guide."*

Tracking Role Authorizations and Members

Query the system tables to determine the task authorizations, object privileges, and roles that each user has been granted. The following table lists each system table that you might want to query.

System Table	Information
RBW_ROLES	Roles that exist in the database
RBW_ROLE_MEMBERS	Members of each role
RBW_USERAUTH	Task authorizations of each user and role
RBW_TABAUTH	Object privileges of each user and role

The following examples illustrate how to query the system tables to determine the access rights within your database.

Examples

The following statement returns a list of all user-created roles in the database:

```
select name, creator
from rbw_roles
order by name ;

NAME                    CREATOR
DATABASE_MANAGEMENT     SYSTEM
MARKETING               SYSTEM
OBJECT_MANAGEMENT       SYSTEM
SECURITY_MANAGEMENT     SYSTEM
TABLE_SELECT            SYSTEM
```

The following statement returns a list of all roles in the database with members. The **USERNAME** column shows the users and roles that are members of the roles listed in the **ROLENAME** column. The **INDIRECT** column shows whether the user or role is an indirect member (Y) or direct member (N).

```
select rolename, username, indirect
from rbw_role_members
order by rolename, username ;

ROLENAME                USERNAME        INDI
DATABASE_MANAGEMEN      JOHN              N
MARKETING               CODY              N
MARKETING               NASI              N
MARKETING               SUDHIR            N
OBJECT_MANAGEMENT       CODY              Y
OBJECT_MANAGEMENT       MARKETING         N
OBJECT_MANAGEMENT       NASI              Y
OBJECT_MANAGEMENT       SUDHIR            Y
SECURITY_MANAGEMEN      CHRIS             N
SECURITY_MANAGEMEN      JUDY              N
```

The following statement returns a list of all users and roles in the database with task authorizations. It also lists each task authorization and shows whether the user or role has the task authorization directly (Y), as a direct member of a role (R), or as an indirect member of a role (I).

```
select grantee, dbaauth, resauth, user_management,
   grant_table, role_management, alter_any, public_macros,
   access_any, modify_any, drop_any, create_any,
   lock_database, upgrade_database,
   alter_table_into_any,create_own, alter_own, grant_own,
   isrole
from rbw_userauth
order by grantee ;

GRANTE       DBAA RESA USER GRAN ROLE ALTE PUBL ACCE MODI DROP  CREA LOCK
REST UPGR ALTE CREA ALTE  GRAN ISRO
CHRIS         N  N  R  R  R  N  N  N  N  N  N  N  N  N  N  N  N   N  N
CODY          N  N  N  N  N  I  I  I  I  I  I  N  N  N  N  N  N   N  N
DATABASE_M    N  N  N  N  N  N  N  N  N  N  N  Y  Y  Y  N  N  N   N  Y
JOE           N  N  N  N  N  N  N  N  N  N  N  N  Y  Y  N  N  N   N  N
JOHN          N  N  N  N  N  N  N  N  N  N  N  R  Y  Y  N  N  N   N  N
JUDY          N  N  R  R  R  N  N  N  N  N  N  N  N  N  N  N  N   N  N
MARIA         N  N  N  N  N  N  N  N  N  N  N  N  Y  Y  N  N  N   N  N
MARKETING     N  N  N  N  N  R  R  R  R  R  R  N  N  N  N  N  N   N  Y
NASI          N  N  N  N  N  I  I  I  I  I  I  N  N  N  N  N  N   N  N
OBJECT_MAN    N  N  N  N  N  Y  Y  Y  Y  Y  Y  N  N  N  N  N  N   N  Y
SECURITY_M    N  N  Y  Y  Y  N  N  N  N  N  N  N  N  N  N  N  N   N  Y
SUDHIR        N  N  N  N  N  I  I  I  I  I  I  N  N  N  N  N  N   N  N
SYSTEM        Y  N  Y  Y  Y  Y  Y  Y  Y  Y  Y  Y  Y  Y  Y  Y  Y   Y  N
TABLE_SELE    N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N   N  Y
```

The **DBA_AUTH** and **RES_AUTH** columns show whether the user or role has been granted the DBA or RESOURCE system roles. The **ISROLE** column shows whether the grantee is a role (Y) or a user (N).

Each task authorization column contains one of the following values.

Value	Meaning
N	User or role does not have the task authorization.
Y	User or role has the task authorization directly.
R	User or role has the task authorization through direct membership in a role.
I	User or role has the task authorization through indirect membership in a role. In other words, the user is a member of a role that has been granted a role with the task authorization.

The following statement returns a list of all users and roles with object privileges in the database:

```
select grantee, grantor, tname, selauth, insauth, delauth,
   updauth
from rbw_tabauth
order by grantee ;

GRANTEE         GRANTOR     TNAME      SELA  INSA  DELA   UPDA
MARIA           SYSTEM      PRODUCT    N     Y     N      N
TABLE_SELECT    SYSTEM      SALES      Y     N     N      N
TABLE_SELECT    SYSTEM      PERIOD     Y     N     N      N
TABLE_SELECT    SYSTEM      PRODUCT    Y     N     N      N
TABLE_SELECT    SYSTEM      MARKET     Y     N     N      N
```

Administering Password Security

Password security features allow you, as the database administrator (or any user with the USER_MANAGEMENT task authorization), to control the longevity and content of database passwords. These features are controlled by password parameters located in the **rbw.config** file. These parameters can set up:

- A password expiration period to control the age of passwords
- A warning message period to notify users of an impending password expiration
- A restriction on the re-creation of old passwords to prevent users from repeatedly using the same password
- A restriction on the frequency of password changes
- The required complexity and length of valid passwords
- A restriction on the number of times users can consecutively attempt to connect to the database without success

The database administrator can implement the appropriate password security features by setting only the parameters that apply to that site or database environment.

The database administrator (or any user with the USER_MANAGEMENT task authorization) initially creates passwords for users by using the GRANT CONNECT statement. To continually use the database, users must comply with the configured password parameters, using the GRANT CONNECT statement to change their own passwords as required. For the syntax of the GRANT CONNECT statement, refer to the *SQL Reference Guide*.

The following table lists the password parameters and describes their functions.

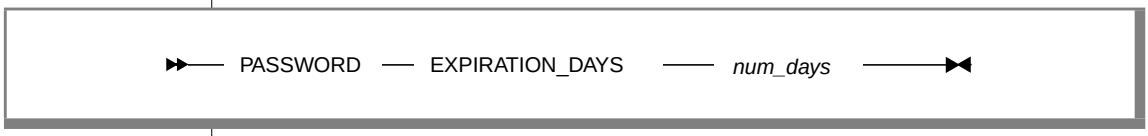
PASSWORD Parameter	Description
EXPIRATION_DAYS	Maximum number of days passwords exist. Forces users to change their passwords regularly.
EXPIRATION_WARNING_DAYS	The number of days prior to password expiration that the user receives a warning message.
RESTRICT_PREVIOUS	Minimum number of password changes that must occur before a password can be reused. Forces users to use different passwords.
CHANGE_MINIMUM_DAYS	Minimum number of days a password must exist before the user can change it. Prevents users from changing their passwords multiple times in quick succession in order to bypass the parameter PASSWORD RESTRICT_PREVIOUS.
MINIMUM_LENGTH	Minimum number of total characters required in each password.
COMPLEX_NUM_ALPHA	Minimum number of alphabetic characters (A to Z, a to z) required in each password.
COMPLEX_NUM_NUMERICS	Minimum number of numeric characters (0 to 9) required in each password.
COMPLEX_NUM_PUNCTUATION	Minimum number of punctuation characters (for example, !@#$%&) required in each password.
LOCK_FAILED_ATTEMPTS	Maximum number of failed connection attempts allowed before a user's account is locked.
LOCK_PERIOD_HOURS	Number of hours a user's account is locked following failed connection attempts.

Enforcing Password Changes

You can force users to change their passwords periodically by setting the PASSWORD EXPIRATION_DAYS parameter in the **rbw.config** file. This parameter sets the maximum number of days passwords can exist before user accounts expire. To avoid password expiration, users must change their passwords with the GRANT CONNECT statement within the specified number of days.

Syntax

To specify a password expiration period, enter a line in the *rbw.config* file with the following syntax.



num_days Specifies the number of days passwords can exist before they expire. This value must be an integer in the range of 0 to 512. The default is `0`, which sets no restriction. Users need never change their passwords. To compute a password expiration date, the *num_days* value is added to the base value stored in the **PASSWORD_TS** column in the **RBW_USERAUTH** system table.

Usage Notes

- Users can change their passwords at any time before the expiration date unless the PASSWORD CHANGE_MINIMUM_DAYS parameter is set, which sets the minimum number of days that must pass before users can change their passwords. If PASSWORD EXPIRATION_DAYS and PASSWORD CHANGE_MINIMUM_DAYS are both set, users must change their passwords after the minimum number of days have passed and before the expiration date. For more information about PASSWORD CHANGE_MINIMUM_DAYS, refer to "Limiting Frequency of Password Changes" on page 7-32.
- If an account expires, the user cannot connect to the database until the database administrator (or any user with the task authorization USER_MANAGEMENT) assigns a new password with the GRANT CONNECT statement.
- When an account expires, the user's status changes from valid to expired, as indicated in the **RBW_USERAUTH** system table. After the database administrator has assigned a new password, the user's status reverts to valid. You can check the status of a user in the **RBW_USERAUTH** table as follows:

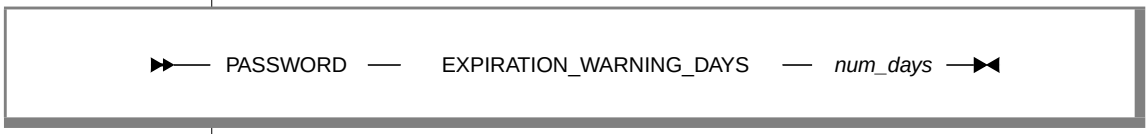
```
select grantee, expired
from rbw_userauth
where grantee = 'user_name' ;
```

Warning Users of Password Expiration

You can warn users that they must change their passwords before the expiration date by setting the PASSWORD EXPIRATION_WARNING_DAYS parameter. This parameter sets the number of days before the password expiration date that users receive a warning message. (The message appears each time they connect to the database.)

Syntax

To specify a password warning period, enter a line in the **rbw.config** file with the following syntax.



num_days Specifies the number of days before password expiration that users receive a warning message. This value must be an integer in the range of 0 to 512 days. The default is `0`, which indicates no warning message.

Usage Notes

- The value of the parameter PASSWORD EXPIRATION_DAYS must be greater than the value of the parameter PASSWORD EXPIRATION_WARNING_DAYS. If not, the parameter PASSWORD EXPIRATION_WARNING_DAYS is ignored, and the expiration period value becomes the warning value. In this case, users receive a warning message every time they connect to the database.
- If the PASSWORD EXPIRATION_DAYS parameter is not set, the PASSWORD EXPIRATION_WARNING_DAYS parameter is ignored.

Example

Assume the **rbw.config** file contains the following entries:

```
PASSWORD EXPIRATION_DAYS 30
PASSWORD WARNING_DAYS 5
```

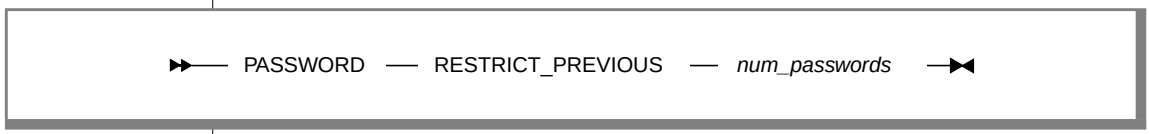
Users must create new passwords at least every 30 days. From the 26th to the 30th day in the life of their current passwords, each time they connect to the database, they receive a warning message that their passwords will expire.

Limiting Reuse of Previous Passwords

You can limit the reuse of previous passwords by setting the PASSWORD RESTRICT_PREVIOUS parameter in the **rbw.config** file. This parameter sets the minimum number of password changes required before users can re-create passwords.

Syntax

To limit the reuse of passwords, enter a line in the **rbw.config** file with the following syntax.



num_passwords Specifies the number of password changes required before users can re-create their passwords. This value must be an integer in the range of 0 to 128. The default value is `0`, which indicates no restrictions for re-creating old passwords. For example, if this parameter is set to `5`, users must change their passwords 5 times before re-creating a password.

Usage Notes

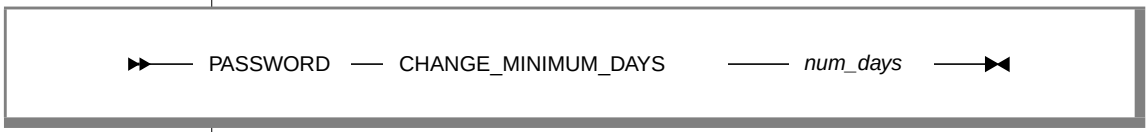
- To prevent users from quickly changing their passwords in order to re-create a password, set the PASSWORD CHANGE_MINIMUM_DAYS parameter.
- The PASSWORD RESTRICT_PREVIOUS restriction applies to the database administrator (and to users with the USER_MANAGEMENT task authorization) only with respect to their own passwords. They can assign a user's previous password to the same user at any time.

Limiting Frequency of Password Changes

You can limit the frequency of password changes by setting the PASSWORD CHANGE_MINIMUM_DAYS parameter in the **rbw.config** file. This parameter sets the number of days that must pass between password changes.

Syntax

To limit the frequency of password changes, enter a line in the **rbw.config** file with the following syntax.



num_days Specifies the number of days that must pass between password changes. This value must be an integer in the range of 0 to 128. The default is `0`, which indicates no restriction on the frequency of password changes.

Example

Assume the **rbw.config** file contains the following entries:

```
PASSWORD EXPIRATION_DAYS 60
PASSWORD RESTRICT_PREVIOUS 5
PASSWORD CHANGE_MINIMUM_DAYS 20
```

Users must create new passwords at least every 60 days. They cannot re-create old passwords until they have created 5 subsequent passwords. After changing their passwords, they must wait 20 days before changing passwords again.

Enforcing Password Complexity and Length

You can force users to create complex and secure passwords by setting any combination of the PASSWORD MINIMUM_LENGTH parameter and the complexity parameters.

The PASSWORD MINIMUM_LENGTH parameter sets the minimum number of characters required in each password, while the complexity parameters set the minimum number of alphabetic, numeric, and punctuation characters. (A punctuation character is any printing character that is not a letter, a number, or a space.) The complexity parameters are as follows:

```
PASSWORD COMPLEX_NUM_ALPHA
PASSWORD COMPLEX_NUM_NUMERICS
PASSWORD COMPLEX_NUM_PUNCTUATION
```

Syntax: MINIMUM_LENGTH

To set the minimum number of characters required in each password, enter a line in the **rbw.config** file with the following syntax.



num_characters Specifies the minimum number of characters required in each password. This value must be an integer in the range of 0 to 128. The default is `0`, which sets no restriction.

Syntax: COMPLEX_NUM_ALPHA

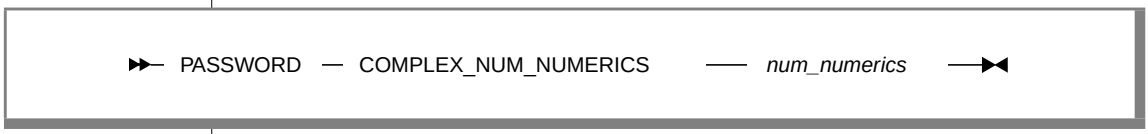
To set the number of alphabetic characters required in each password, enter a line in the **rbw.config** file with the following syntax.



num_alpha Specifies the minimum number of alphabetic characters required in each password. This value must be an integer in the range of 0 to 42. The default is `0`, which sets no restriction.

Syntax: COMPLEX_NUM_NUMERICS

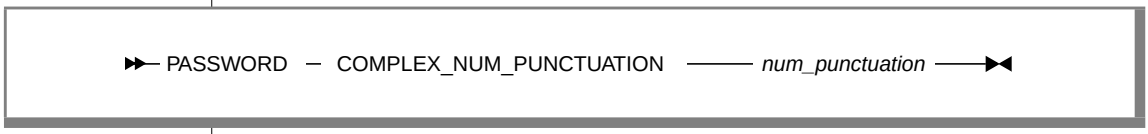
To set the number of numeric characters required in each password, enter a line in the **rbw.config** file with the following syntax.



num_numerics Specifies the minimum number of numeric characters required in each password. This value must be an integer in the range of 0 to 42. The default is `0`, which sets no restriction.

Syntax: COMPLEX_NUM_PUNCTUATION

To set the number of punctuation characters required in each password, enter a line in the **rbw.config** file with the following syntax.



num_punctuation Specifies the minimum number of punctuation characters required in each password. This value must be an integer in the range of 0 to 42. The default is `0`, which sets no restriction.

Usage Notes

- You can set password complexity parameters without also setting the PASSWORD MINIMUM_LENGTH parameter. The combined total of the complexity parameters becomes the minimum required length.
- To enforce a minimum length that is different from the combined total of the complexity parameters, the minimum length parameter must be greater than the combined total. If the combined total exceeds the minimum length value, the minimum length parameter is ignored, and the combined total becomes the minimum required length.

Examples

Suppose the **rbw.config** file contains the following entries:

```
PASSWORD COMPLEX_NUM_ALPHA 4
PASSWORD COMPLEX_NUM_NUMERICS 2
PASSWORD COMPLEX_NUM_PUNCTUATION 2
PASSWORD COMPLEX_MINIMUM_LENGTH 10
```

When users create new passwords, the passwords must have at least 10 characters with at least 4 alphabetic, 2 numeric, and 2 punctuation characters. The following GRANT CONNECT statements create valid passwords:

```
grant connect to craig with 'dbs1are2fun$%' ;
grant connect to james with 'sq67lis%fu*%' ;
```

The following GRANT CONNECT statements return error messages because, given the previous configuration, the passwords are invalid:

```
grant connect to maria with dbuser ;
grant connect to prema with perfor12mance ;
```

Locking User Accounts After Failed Connection Attempts

You can limit the number of times users can incorrectly enter their passwords by setting the PASSWORD LOCK_FAILED_ATTEMPTS parameter. This parameter sets the number of consecutive failed connection attempts allowed before user accounts are locked. The count of failed connection attempts is reset for a given user each time that user successfully connects to the database.

Syntax

To limit the number of times users can incorrectly enter their passwords, enter a line in the **rbw.config** file with the following syntax.



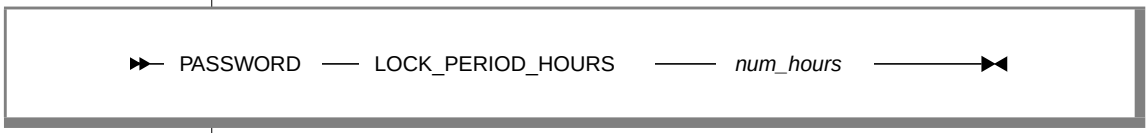
num_attempts Specifies the number of failed connection attempts allowed before user accounts are locked. This value must be an integer in the range of 0 to 128. The default is `0`, which indicates no restriction on the number of incorrect entries.

Specifying the Lock-Out Period

You can specify the duration of a locked account by setting the PASSWORD LOCK_PERIOD_HOURS parameter. This parameter sets the number of hours accounts are locked following a lock-out caused by failed connection attempts. After the configured number of hours passes, users can connect to the database with the same password.

Syntax

To specify the duration of the lock-out period, enter a line in the **rbw.config** file with the following syntax.



num_hours Specifies the number of hours accounts are locked following a lock-out caused by failed connection attempts. This value must be an integer in the range of 0 to 128. The default is `0`, indicating an indefinite lock-out period. If this parameter is set to `0` and an account is locked, the database administrator (or any user with the USER_MANAGEMENT task authorization) must assign a new password.

Locked Account Status

If an account is locked, the user cannot connect to the database until the database administrator has assigned a new password or until the number of hours set in the PASSWORD LOCK_PERIOD_HOURS passes.

When an account is locked, the user's status changes from valid to locked, which is indicated in the **RBW_USERAUTH** system table. After the database administrator assigns a new password, the user's status reverts to valid. You can check the locked status of a user in the **RBW_USERAUTH** table with the following query:

```
select grantee, locked
from rbw_userauth
where grantee = 'user_name' ;
```

Chapter

8

Managing Database Activity in an Enterprise

Task Authorizations for Managing Database Activity 8-4

Administration Database 8-4

Monitoring Database Activity with Dynamic Statistic Tables 8-6
- Read and Write Statistics 8-7
  - Definition of Read Statistics 8-7
  - Definition of Write Statistics 8-9
  - Platform Dependency 8-9

Controlling Database Activity 8-9
- Bringing a Database to a Quiescent State 8-10
- Activating a Database 8-10
- Resetting Accumulated Statistics 8-11
- Canceling a User Command 8-11
- Closing a User Session 8-11
- Changing User Priorities for the Current Session 8-12

Administration Daemon Process 8-12
- Statistics Collection Interval 8-14
- DST Refresh Interval 8-14

Event Logging 8-15
- Logging Subsystem 8-16
  - Log Daemon 8-16
  - Log Viewer 8-17
- Event Log Messages 8-21
  - Event Severity 8-21
  - Event Category 8-22
  - Audit Events 8-22
  - Error Events 8-22
  - Operational Events 8-23

Schema Events 8-23
Usage Events 8-23
Log Files 8-24
Configuring the Logging Subsystem 8-25
Setting the Startup State 8-25
Specifying the Location of Log Files 8-25
Specifying the Maximum Log File Size 8-26
Setting the Log Severity Filter Level 8-27
Controlling Logging Operations. 8-27
Starting Logging 8-28
Stopping Logging. 8-28
Switching Log Files 8-28
Changing Log Filter Levels 8-28
Terminating the Log Daemon. 8-28
Query Logging 8-29

Controlling Advisor Logging 8-29
Advisor Log Files 8-29
What the Advisor Logs 8-30
Starting and Stopping the Advisor Log. 8-31
ADMIN ADVISOR_LOGGING 8-31
ALTER SYSTEM 8-32
ADMIN ADVISOR_LOG_DIRECTORY 8-32
ADMIN ADVISOR_LOG_MAXSIZE 8-33

Accounting 8-34
Accounting Process 8-35
Format of Accounting Records 8-35
Accounting Files 8-37
Configuring Accounting. 8-38
Setting the Startup State 8-38
Specifying the Location of Accounting Files. 8-39
Specifying the Maximum Accounting File Size. 8-39
Setting the Accounting Mode. 8-40
Controlling Accounting 8-40
Starting Accounting 8-41
Stopping Accounting 8-41
Switching Accounting Files 8-41
Changing Accounting Mode 8-41

In This Chapter

This chapter describes the tasks involved in monitoring, controlling, and tracking resource use of Red Brick Decision Server. It includes the following sections:

- Task Authorizations for Managing Database Activity
- Administration Database
- Monitoring Database Activity with Dynamic Statistic Tables
- Controlling Database Activity
- Administration Daemon Process
- Event Logging
- Controlling Advisor Logging

Although enterprise scenarios differ from case to case, they often share some characteristics:

- Multiple Red Brick Decision Server installations on different platforms
- Duplication of data over multiple databases
- Relatively high number of users accessing the database
- Users with varying levels of expertise accessing the database
- Users from different departments accessing the database

Database administration tasks associated with these scenarios include:

- Monitoring and controlling contention for database resources (for example, CPU time, database tables, disk I/O)
- Determining the level of database use by user or department in order to implement a charge-back accounting system

- Copying data between tables in different database installations

  For information on moving data among database servers, refer to the copy management utility, **rb_cm**, in the *Table Management Utility Reference Guide*.

To accomplish these tasks, the database administrator must be able to monitor the database activity of users, their sessions, and the queries issued by those sessions; identify resource contention, misuse of the system, or queries that are consuming too many resources; and perform actions to control database activity.

Task Authorizations for Managing Database Activity

To monitor and control database activity, a user must have two task authorizations: the ACCESS_SYSINFO and ALTER_SYSTEM authorizations. The DBA system role includes these authorizations. For more information on task authorizations, refer to Chapter 7, "Providing Database Access and Security."

Users with the ACCESS_SYSINFO authorization can monitor activity on the database to which they are currently connected. Users with the ALTER_SYSTEM authorization can control the use of the database to which they are currently connected.

Administration Database

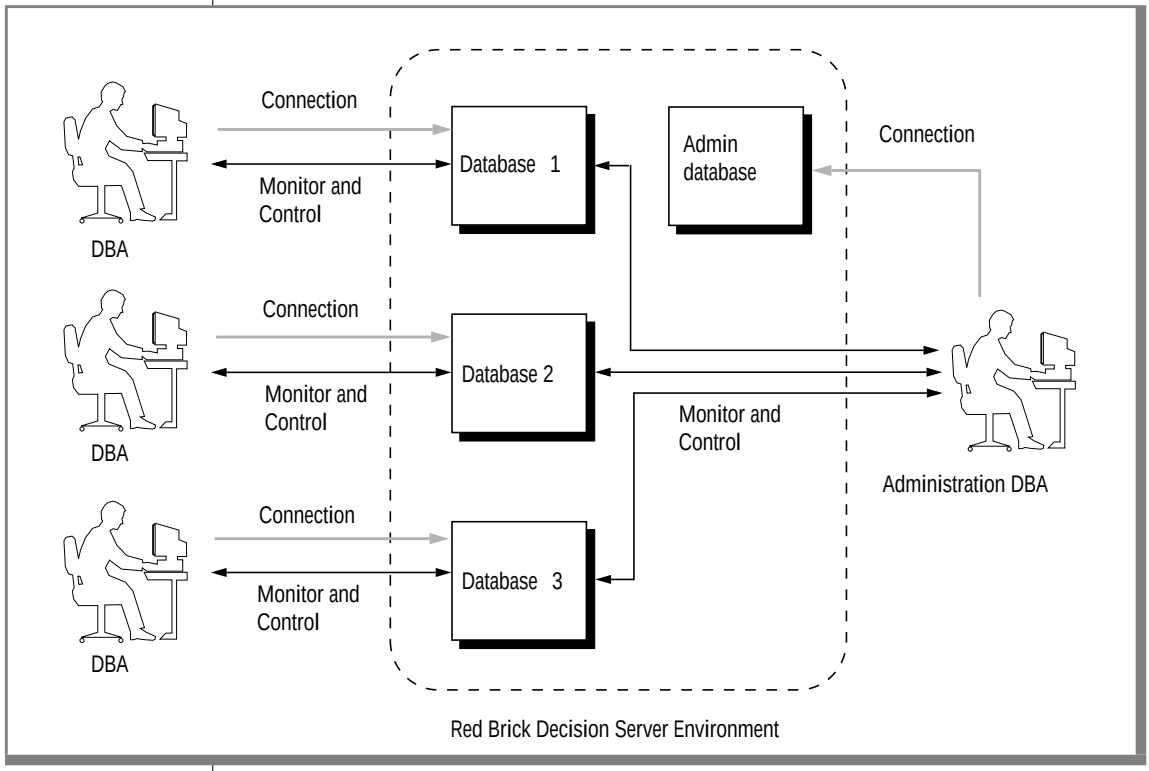
The administration database is used by the database administrator to monitor and control all of the databases within an enterprise. When you install Red Brick Decision Server with the installation script, the script asks whether you want to create the administration database. Informix recommends that you install it. The installation script builds the administration database in a subdirectory of the installation directory named **admin_db**.

As its name suggests, the administration database is for administrative purposes only. The database contains only system tables, and you cannot create any segments or tables in it. It provides the following capabilities:

- A user who has ACCESS_SYSINFO authorization for the administration database and is connected to that database can obtain database activity statistics for *all* databases.
- A user who has ALTER_SYSTEM authorization for the administration database and is connected to that database can perform administrative actions on *all* warehouse databases.

The following figure illustrates the role of the administration database.

Figure 8-1
Role of Administration Database



Monitoring Database Activity with Dynamic Statistic Tables

Statistics on the activity associated with each active database in the database server are available through a set of dynamic statistic tables (DSTs). The database administrator can *monitor* database use by querying the DSTs. DSTs are nonpersistent; that is, they are not stored anywhere on disk but held and periodically updated in memory. Although the DSTs do not exist on disk, they appear as entries in the **RBW_TABLES** system table. These entries allow front-end tools to perform queries against the DSTs. The dynamic statistic tables are as follows:

- **DST_DATABASES**
- **DST_USERS**
- **DST_SESSIONS**
- **DST_COMMANDS**
- **DST_LOCKS**

For column names and descriptions of each table, refer to Appendix C, "System Tables and Dynamic Statistic Tables."

Users with ACCESS_SYSINFO authority on the current database can retrieve the rows in the DSTs that are relevant to that database. Users who have ACCESS_SYSINFO authority on the administration database and are connected to that database can retrieve information from the DSTs on all databases for which the administration daemon has information.

The administration daemon holds information only on databases that have been accessed at least once since the administration daemon was started. Similarly, the administration daemon holds information only on users who have accessed a database at least once since the administration daemon was started. For more information on the administration daemon, refer to "Administration Daemon Process" on page 8-12.



Tip: *You can define views on the DSTs consisting of useful subsets of the table columns.*

Read and Write Statistics

Many of the DSTs contain the following I/O statistics:

- Cache reads and writes
- Logical reads and writes
- Physical reads and writes

These statistics are important but easily misinterpreted. Therefore, they are defined here.

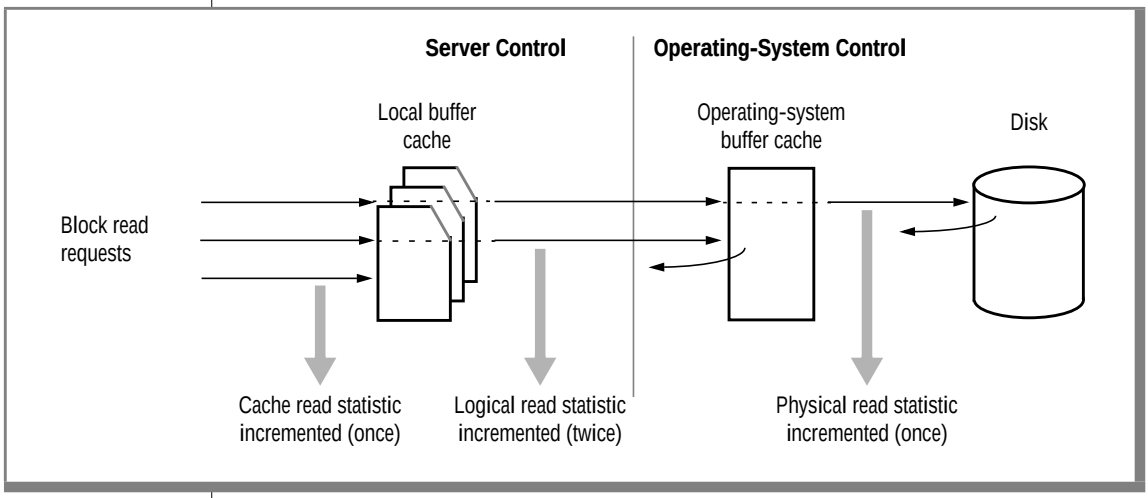
Definition of Read Statistics

Consider the case in which a session must read data. It first determines the location of that data (the particular block) and attempts to lock the block. The database server process checks that the required block is already in the local buffer cache. If the block is there (that is, if the data has already been read into the local buffer cache for use by the session), the block is locked, and the cache read statistic is incremented by one. Subsequent reads of individual rows in the locked block do not affect the cache read statistic because this statistic counts only *block* read requests. If the required block is not already in the local buffer cache, however, the session issues a logical read request, and the logical read statistic is incremented.

A logical read is a call to the operating system to read a block of data. If the data is in an operating-system buffer, no physical read to disk is required. If the data is not in an operating-system buffer, the operating system performs a read from disk, and the physical read statistic is incremented.

The following figure illustrates how cache read, logical read, and physical read statistics are generated.

Figure 8-2
How Read Statistics are Generated



The sum of the cache reads and the logical reads is the total number of block read requests by the session. The ratio of the cache read requests to the total number of block read requests represents the ratio of cache buffer hits.

Tip: *System table reads bypass the local buffer cache and are not reported. However, these reads represent a very small fraction of the total, so they should not significantly affect the overall cache hit rate.*

Definition of Write Statistics

When a process needs to perform a write, it attempts to lock the appropriate block in memory. Consider the following cases:

- The block is not in the local buffer cache. In this case, the database server must read the block from disk. The database server process can then lock the block for writes. When the database server finishes writing to this block, it must eventually write the block back to disk, so the logical write statistic is incremented by one.
- The block is in the local buffer cache and has not been written to. In this case, the database server process can lock the block for writes. Again the database server must eventually write the block back to disk, so the logical write statistic is incremented by one.
- The block is in the local buffer cache and is dirty. That is, the block has already been written to. The database server process can lock the block to perform additional writes. Because the logical write statistic was incremented for the original writes, this statistic is not incremented. Instead the cache write statistic is incremented by one.

Platform Dependency

When a logical read or write is performed, the operating system usually performs a corresponding physical read or write to disk. However, the operating system keeps the physical I/O data, so the availability of these statistics is platform dependent.

Controlling Database Activity

To control database activity, use the ALTER SYSTEM statement. Users with ALTER_SYSTEM authorization can execute the ALTER SYSTEM statement to control use of the current database. Users with ALTER_SYSTEM authorization on the administration database who are connected to the administration database can use the ALTER SYSTEM statement to control use of *all* databases in the warehouse.

With the ALTER SYSTEM statement, a user with the necessary authority can perform the following operations:

- Bring a database to a quiescent state
- Activate a database
- Reset accumulated statistics
- Cancel a user command
- Close a user session
- Change user priority for current sessions

The following sections describe all of these operations. For a complete description of the ALTER SYSTEM syntax, refer to the *SQL Reference Guide*.

Bringing a Database to a Quiescent State

You can use the ALTER SYSTEM QUIESCE DATABASE statement to bring a database to a quiescent state. In this state, the database does not allow any new sessions or any new commands for existing sessions, but currently executing commands are allowed to complete.

Use this statement as preparation for shutting down the **rbwapid** daemon process. The quiescent state is also useful for performing maintenance tasks such as disk drive maintenance.



Important: *A user with the IGNORE_QUIESCE task authorization can perform actions on a quiescent database, thus overriding another user's ALTER SYSTEM QUIESCE DATABASE statement. All users with the DBA system role automatically have the IGNORE_QUIESCE task authorization.*

Activating a Database

You can use the ALTER SYSTEM RESUME DATABASE statement to bring a database to normal working mode.

The RESUME DATABASE clause must be issued by an existing session (because you cannot start a new session on a quiescent database) or by a user who is connected to the administration database and has ALTER SYSTEM authorization for that database.

Resetting Accumulated Statistics

You can use the ALTER SYSTEM RESET STATISTICS statement to reset all the DST statistics for a database to zero.

Canceling a User Command

You can use an ALTER SYSTEM CANCEL USER COMMAND statement to cancel the currently executing statement for a specific session. You can also cancel the currently executing statements for:

- All sessions for a specific user on a specific database
- All sessions for all users on a specific database

If you have the ALTER_SYSTEM authority on the administration database, you can cancel the currently executing statements for:

- All sessions for a specific user on all databases
- All sessions for all users on all databases

If a session is not executing a command, the ALTER SYSTEM CANCEL COMMAND statement is ignored.

Closing a User Session

You can use the ALTER SYSTEM CLOSE USER SESSION statement to terminate a specific session. You can also terminate:

- All sessions for a specific user on a specific database
- All sessions for all users on a specific database

If you have the ALTER_SYSTEM authority on the administration database, you can terminate:

- All sessions for a specific user on all databases
- All sessions for all users on all databases

When you use this option to terminate a session, any statements that the session is currently executing are canceled, and a message is sent to the session stating that the session was terminated by an operator action.

Changing User Priorities for the Current Session

You can use the ALTER SYSTEM CHANGE USER PRIORITY statement to change the priority of a specific session. You can also change the priorities of:

- All sessions for a specific user on a specific database
- All sessions for all users on a specific database

If you have the ALTER_SYSTEM authority on the administration database, you can change the priorities of:

- All sessions for a specific user on all databases
- All sessions for all users on all databases

Changes to user priority take place immediately for the sessions and show up in the PRIORITY column of the DST_SESSIONS table. These changes are not permanent, however. Any new sessions started for the user have the original priority. To make a permanent change to a user priority, use the ALTER USER statement described in the *SQL Reference Guide*.

UNIX

Your platform must have the UNIX **renice** command in order to support user priorities. You must specify the full pathname of the **renice** executable file with the ADMIN RENICE_COMMAND configuration parameter. ♦

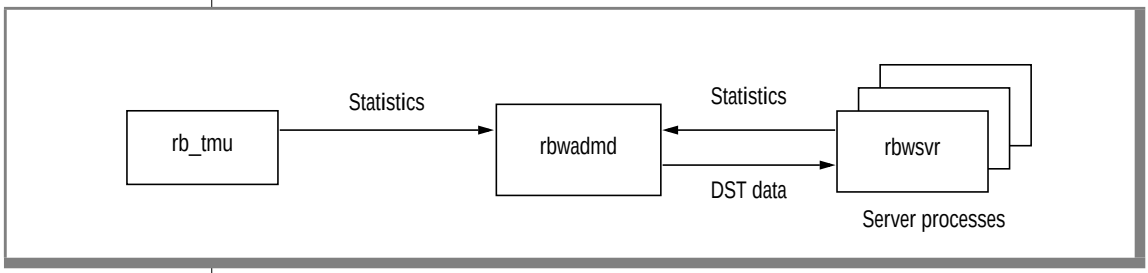
Administration Daemon Process

The administration daemon process collects statistics for the DSTs and executes ALTER SYSTEM statements. The administration daemon process (**rbwadmd**) is started at the same time as the warehouse (**rbwapid**) and log (**rbwlogd**) daemon processes.

For information on restarting the administration daemon if it goes down, refer to "Monitoring and Controlling a Database Server on UNIX" on page 9-59 or "Monitoring and Controlling a Database Server on Windows" on page 9-62.

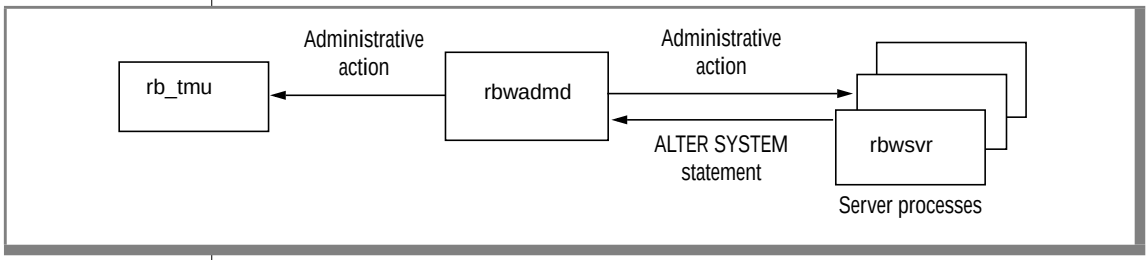
The administration daemon collects statistics from the TMU and database server processes and returns DST data to a database server process. The following figure illustrates statistics data flow between the administration daemon and the other database processes.

Figure 8-3
Data Flow Between rbwadmd and Other Database Processes



The administration daemon accepts ALTER SYSTEM statements from the database server processes and performs the appropriate administrative actions on the TMU or on another database server process. The following figure illustrates the ALTER SYSTEM flow of control between the administration daemon and the other database server processes.

Figure 8-4
ALTER SYSTEM Flow of Control



Statistics Collection Interval

Collection of database activity statistics begins as soon as the administration daemon is started and continues for as long as the process is running. Unless you specifically reset the statistics for a database, the collection interval for that database is the interval that the administration daemon has been running.

If you terminate the administration daemon, you lose all statistics currently held in the dynamic statistic tables.

The administration daemon collects statistics only for databases that have been accessed at least once since the administration daemon on UNIX or Red Brick Decision Server service on Windows was started. Similarly, the administration daemon collects statistics only for database users who accessed a database at least once after the administration daemon starts.

To reset all statistics for a database (or all databases) to zero, use the ALTER SYSTEM RESET STATISTICS statement.

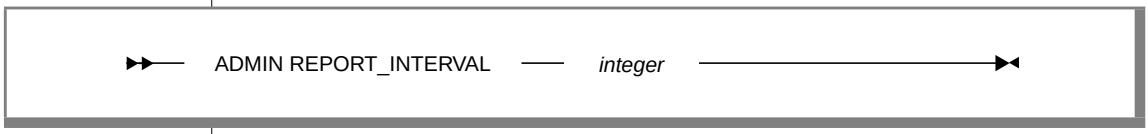
DST Refresh Interval

To set the maximum interval between dynamic statistic table refreshes, use the ADMIN REPORT_INTERVAL configuration parameter and the SET REPORT_INTERVAL command. This interval is accurate to +/- 1 minute and might be less accurate with parallel processing.

Whenever a statement requires a change of state or whenever a session requests, acquires, or releases a lock, the database server sends updates to the dynamic statistic tables. The states of a statement include connecting, idle, executing, compiling, calculating, returning rows, sorting, building indexes, and inserting. If the time between such events exceeds the value that you specify for the configuration parameter ADMIN REPORT_INTERVAL, the dynamic statistic tables are automatically refreshed. To override the ADMIN REPORT_INTERVAL value for the duration of a session, use the SET REPORT_INTERVAL command.

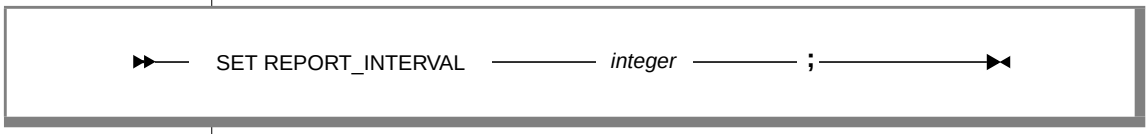
Syntax

To set ADMIN REPORT_INTERVAL, add an entry in the **rbw.config** file. The syntax for setting this parameter is as follows.



The *integer* value indicates the interval in minutes.

To set the DST refresh interval for a session, issue a SET REPORT_INTERVAL command using the following syntax.



To turn off statistics collection, set the DST refresh interval to zero (either by setting the configuration parameter or issuing a SET REPORT_INTERVAL command for a session).

Event Logging

In enterprise systems, many users from different departments often access the same Red Brick Decision Server databases. In this environment, it is helpful to have a record of system events such as user activities, operational events, and audit events. The event-logging feature provides such information. This information allows you to determine whether the system is being used correctly and helps you to diagnose error conditions.

The event logging feature generates records for a wide range of database server events (audit events, error conditions, administrative actions, schema changes, and end-user operations) and stores the records on disk. You can display these log records as they are generated, or you can display all the log records generated over some interval (for example, a day) to analyze the recent system history.

Logging Subsystem

Event logging is handled by a separate subsystem: the logging subsystem. The logging subsystem consists of a log daemon and a log viewer.

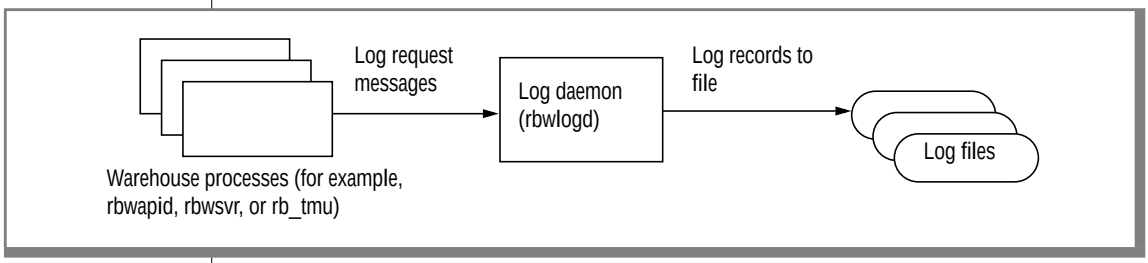
Log Daemon

The log daemon process (**rbwlogd**) handles log request messages issued by Red Brick Decision Server processes when various events occur.

On UNIX, the log daemon is started by the warehouse daemon (**rbwapid**) when you start that process. On Windows, the log process is started by the warehouse thread (**rbw.exe**) when you start the Red Brick Decision Server service. Any database server process can send log request messages to the log daemon.

For example, the **rb_tmu** process might generate a load-initiated message, and the **rbwapid** process might generate a message when an abnormal database server exit occurs. When a warehouse process generates a log request message, it does not wait for a response. To minimize performance impact, all communication between the database server and the log daemon is one-way. For example, when a user drops a table, the database server process for that user sends a message to the log daemon and then continues its processing. The following figure illustrates the role of the log daemon.

Figure 8-5
Log Daemon

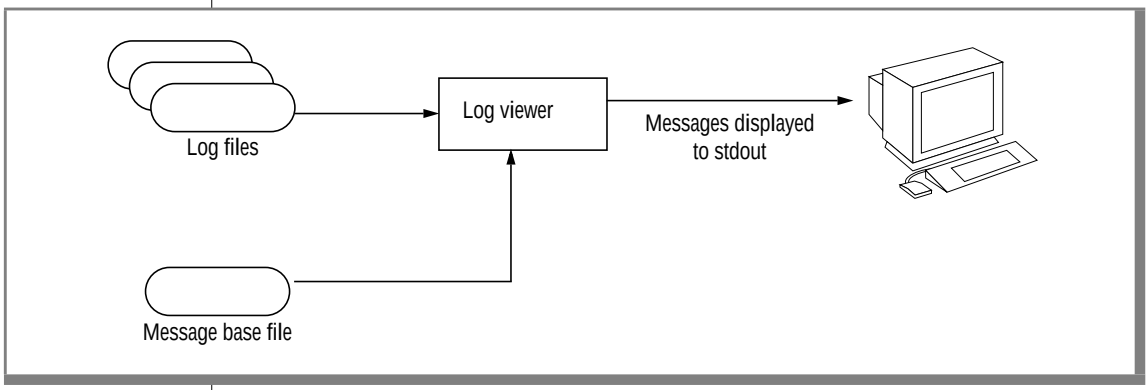


When the log daemon receives a log request message from a database server process, it adds a time stamp and writes the information contained in that message to a log file. Only one log file is active at any given time. If you are not running event logging, no log file is active. When the disk space limits specified by the ADMIN LOG_MAXSIZE configuration parameter are reached, the log daemon closes the file and initializes a new file. For more information, refer to "Log Files" on page 8-24.

Log Viewer

You can use the log viewer utility to view the contents of the log. The log daemon writes the parameter values contained in a log message to the log file. The log daemon does not write full message text to the log file. This text is stored in the form of message templates in a separate message base file. When you view the event messages, the log viewer combines the appropriate message template with the parameter values stored in the log file to give you a readable output. The resulting message is displayed to **stdout** or written to a file. For a discussion of the message parameters, refer to "Event Log Messages" on page 8-21.

Figure 8-6
The Log Viewer



The log viewer executable is named **rbwlogview** on UNIX and **logdview** on Windows. Any user who has read permission for the log files can view event messages with the log viewer. The log files are owned by the **redbrick** account.

The commands have the following syntax:

UNIX

```
rbwlogview [-a] [-t] [-e] [-f] [-p pid]
   [-d database [[-d database] ...]]]
   [-i sourceid [[-i sourceid] ...]]]
   [-c [a][e][o][s][u]]
   [-s [a][r][u]]
   [logfile [[logfile] ...]]
```

♦

WIN NT/2000

```
logdview [-a] [-t] [-e] [-f] [-p pid]
   [-d database [[-d database] ...]]]
   [-i sourceid [[-i sourceid] ...]]]
   [-c [a][e][o][s][u]]
   [-s [a][r][u]]
   [logfile [[logfile] ...]]
```

♦

Option	Description
-a	Specifies the active log file. If you use this option, you cannot specify *logfile*.
-t	Specifies terse output with shorter headers.
-e	Specifies continuous display of the active log file. As new records are written to the active file, they are displayed to **stdout**. (Similar to the UNIX **tail** command.)
-f	Specifies continuous display of active log file. All records currently in the active file are first displayed, followed by any new records as they are written to file. (Similar to the UNIX **tail** command.)
-p *pid*	Displays only log records that originate from the process with the specified process ID
-d *database*	Displays only log records generated by process accessing the specified database. Multiple databases can be specified
-i *sourceid*	Displays only log records that originate from process with the specified *sourceid* (for example, **rb_tmu**).

(1 of 2)

Option	Description
-c	Limits display to the specified categories, one or more of the following: **a** (audit), **e** (error), **o** (operational), **s** (schema), **u** (usage). For more information on event categories, refer to "Event Category" on page 8-22.
-s	Limits display to the specified severities, one or more of the following: **u** (urgent), **a** (alert), **r** (routine). For more information on event severities, refer to "Event Severity" on page 8-21.
logfile	Specifies a particular log file to read from. Multiple files can be specified and are read in the order that they were saved by the log daemon. For more information on log files, refer to "Log Files" on page 8-24.

(2 of 2)

Usage

You must specify either one or more closed log files or the active log file as log viewer input. Otherwise, the log viewer reads from **stdin**. The **-a** option causes the log viewer to read from the active log file. You cannot specify both a closed log file and the **-a** option.

The log viewer program writes to **stdout**, so you can use it in conjunction with a front-end interface. For example, you can redirect the output from the following command to a program with a graphical user interface.

Operating System	Command
UNIX	rbwlogview -a -e
Windows	logdview -a -e

UNIX

Example

The following command displays log records in the closed log file **rbwlog.HOST.950621.101053**. Only log records generated for the **SUPPORT** database and belonging to the SCHEMA category are displayed.

```
% rbwlogview -d SUPPORT -c s rbwlog.HOST.950621.101053
Jun 21 08:59:12 rbwsvr[20158] DB:SUPPORT SCH300R: CREATE TABLE D1
completed successfully.
Jun 21 08:59:13 rbwsvr[20158] DB:SUPPORT SCH300R: CREATE SYNONYM
S2 completed successfully.
...
```

The next command displays records for all logged database server events as they are logged.

```
% rbwlogview -a -e
Jun 21 15:50:46 rbwapid[20500] OPE077R: Server process PID:6603
started for userid:"redbrick"
Jun 21 15:50:59 rbwsvr[6603] DB:AROMA_DB SCH300R: CREATE TABLE
SNOOZE completed successfully.
```

WIN NT/2000

Example

The following command displays log records in the closed log file **rbwlog.HOST.950621.101053**. Only log records generated for the **SUPPORT** database and belonging to the SCHEMA category are displayed.

```
c:\> logdview -d SUPPORT -c s rbwlog.HOST.950621.101053
Jun 21 08:59:12 rbwsvr[20158] DB:SUPPORT SCH300R: CREATE TABLE D1
completed successfully.
Jun 21 08:59:13 rbwsvr[20158] DB:SUPPORT SCH300R: CREATE SYNONYM
S2 completed successfully.
...
```

The next command displays records for all logged database server events as they are logged.

```
c:\> logdview -a -e
Jun 21 15:50:46 rbwapid[20500] OPE077R: Server process PID:6603
started for userid:"redbrick"
Jun 21 15:50:59 rbwsvr[6603] DB:AROMA_DB SCH300R: CREATE TABLE
SNOOZE completed successfully.
```

Event Log Messages

All log messages issued by database server processes have the following parameters:

- Message category
- Message number
- Message severity level

A log message might have additional parameters specific to it. For example, if a user drops a table, the database server processes generates a message with a message category, message number, message severity, and additional *table_name* parameter. The log processes write these parameters to the active log file as a variable-length record.

When viewed using the log viewer, all messages have the following form:

```
CCC###S: Message Text
```

The first three characters (*CCC*) indicate the message category, the next three numeric characters (*###*) represent the message number, and the seventh character (*S*) indicates the message severity. The message text follows the seven-character code, as in the following example:

```
OPE081A: New accounting level is WORKLOAD
```

Event Severity

Each event has one of the following event severity levels:

- ROUTINE
- ALERT
- URGENT

The lowest severity is ROUTINE, and the highest severity is URGENT.

Event Category

Logged events fall into the following categories:

- AUDIT
- ERROR
- OPERATIONAL
- SCHEMA
- USAGE

You can specify a minimum severity level for logged records in each log category. For example, you can specify that the minimum severity for ERROR events is ALERT. In this case, only ERROR events with ALERT or URGENT severity are logged. To specify the minimum severity level for an event category, either use the ALTER SYSTEM CHANGE LOGGING LEVEL statement or set the appropriate configuration parameter directly. For more information on setting the log severity level, refer to "Setting the Log Severity Filter Level" on page 8-27.

Audit Events

Audit events are related to security and access control. Changes to roles, access permissions, password protection, and so on generate AUDIT event records. The default minimum severity level for audit records is ALERT.

An example of an AUDIT event record is:

```
AUD011A: User smith supplied incorrect password.
```

Error Events

Error events are user actions or changes in the database server system environment that cause errors or exceptions. The default minimum severity level for error records is ROUTINE.

An example of an ERROR event record is as follows:

```
ERR717R: Pipe command not allowed with tape output.
```

Operational Events

Operational events are administrative actions such as component startup or shutdown and changes to operational states taken by database administrators (or other users who are members of the DBA system role). The default minimum severity level is ALERT.

An example of an OPERATIONAL event record is as follows:

```
OPE081A: New accounting level is WORKLOAD.
```

Schema Events

Schema events are either changes to physical database structures (creating and dropping PSUs and segments) or changes to logical database structures (all DDL statements). The default minimum severity level is ROUTINE.

An example of a SCHEMA event record is as follows:

```
SCH300R: CREATE TABLE SALES completed successfully.
```

Usage Events

Usage events are end-user operations in the statement system including LOAD DATA, UNLOAD, and all DML statements (for example, SELECT statements). The default minimum severity level is ALERT.

An example of a USAGE event record is as follows:

```
USA302R: LOAD DATA into SALES completed.
```

Setting the USAGE ROUTINE event logs all SQL queries that Red Brick Decision Server processes. This setting is useful in debugging problems with query tools. This setting logs all SQL queries and, depending on system usage, can cause the log files to grow rapidly.

Log Files

The log daemon writes log records to the active log file. When this file exceeds the size specified by the ADMIN LOG_MAXSIZE configuration parameter, the log daemon closes the file and creates a new active file. The log daemon also closes the active file when you stop the logging daemon, and it creates a new active file upon logging startup. In this manner, a sequence of log files accumulates over time. The contents of the files in this sequence can be concatenated before processing because none of the files contain any header or trailer information.

The active log file and the saved log files have the following naming conventions.

Operating System	Command
UNIX	rbwlog.<*daemon_name*>.active
	rbwlog.<*daemon_name*>.<*datetime_stamp*>
Windows	rbwlog.<*service_name*>.active
	rbwlog.<*service_name*>.<*datetime_stamp*>

The ***<datetime_stamp>*** suffix on saved filenames indicates the date and time at which the log daemon closed the file.

The default location for these files is the **$RB_CONFIG/logs** directory on UNIX or the **%RB_CONFIG%\logs** directory on Windows. To specify a different location, set the ADMIN LOG_DIRECTORY configuration parameter. All log files are owned by the **redbrick** account.

Old log files are not removed automatically. The statement administrator must provide a script to periodically remove these files or remove them manually.



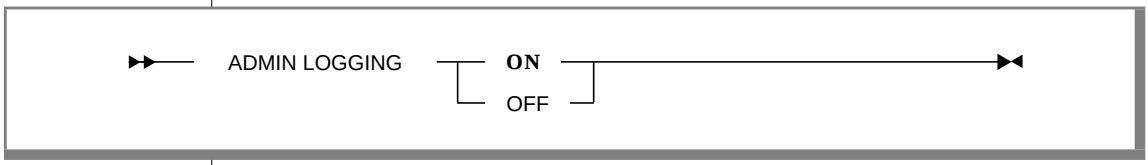
Warning: *If you do not remove old log files, these files accumulate over time and potentially consume all the free disk space. In addition, if you do not specify a value for ADMIN LOG_MAXSIZE, the log daemon writes to a single log file that grows until limited by available disk space.*

Configuring the Logging Subsystem

To configure the logging subsystem, set various configuration parameters in the **rbw.config** file.

Setting the Startup State

The ADMIN LOGGING parameter determines the startup actions of the logging daemon. The syntax for setting this parameter is as follows.



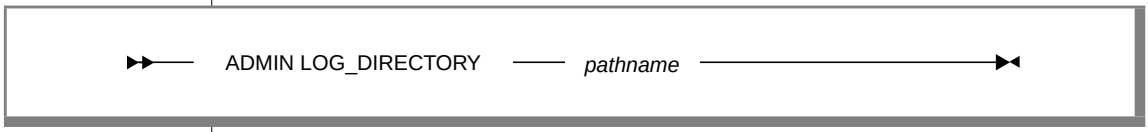
If the ADMIN LOGGING parameter is set to ON, the log daemon creates an initial log file when it initializes and starts logging events. If this parameter is set to OFF, the log daemon does not perform any operations.



Tip: *To start or stop logging while the log daemon is running, use the ALTER SYSTEM START LOGGING and ALTER SYSTEM STOP LOGGING statements.*

Specifying the Location of Log Files

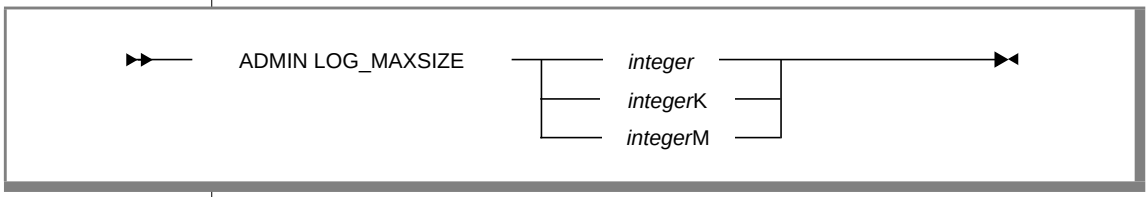
The ADMIN LOG_DIRECTORY parameter specifies the directory in which the log daemon creates the log files. The syntax for setting this parameter is as follows.



The *pathname* variable can be an absolute pathname or a relative pathname. Relative pathnames are interpreted relative to the directory specified by the **RB_CONFIG** environment variable. If you do not set the ADMIN LOG_DIRECTORY parameter, the default logging directory is **$RB_CONFIG/logs** on UNIX or **%RB_CONFIG%\logs** on Windows.

Specifying the Maximum Log File Size

The ADMIN LOG_MAXSIZE parameter specifies the maximum log file size. The syntax for setting this parameter is as follows.



When a log file exceeds the size that this parameter specifies, the log daemon closes the file and creates a new active file in the same directory. The units of the integer value are interpreted as follows:

- Bytes if neither K nor M is specified
- Kilobytes (1024 bytes) if K is specified
- Megabytes (1,048,576 bytes) if M is specified

If you specify K or M, this suffix must immediately follow the numeric value with no spaces.

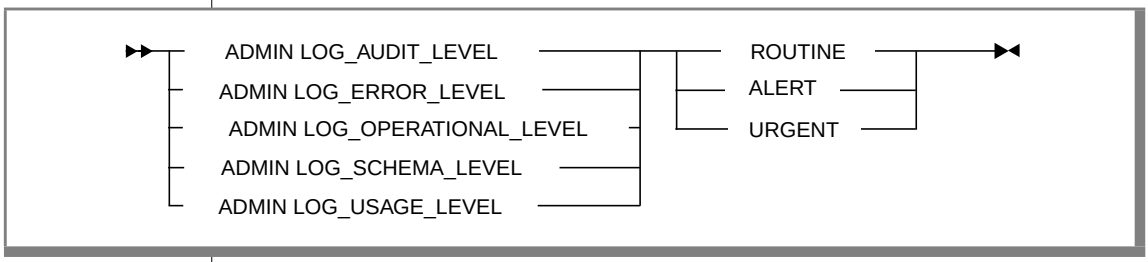
The minimum value for this parameter is 10 kilobytes (10,240 bytes).



Warning: *If you do not set this configuration parameter or if you set it to zero or a negative number, no maximum size is imposed on log files. In this case, log files can continue to grow, limited only by available disk space in the log directory.*

Setting the Log Severity Filter Level

Separate configuration parameters set the log severity filter level, one for each message category. The log severity filter level represents the minimum severity an event within a given category must have in order to be logged. The syntax for these parameters is as follows.





Tip: *To change the filtering level for each logging category during warehouse operation, use the ALTER SYSTEM CHANGE LOGGING LEVEL statement.*

Controlling Logging Operations

To control logging operations, use the ALTER SYSTEM statement. This statement has options for:

- Starting logging
- Stopping logging
- Switching log files
- Changing log filter levels
- Terminating the log daemon



Tip: *The log daemon must be running in order to execute ALTER SYSTEM statements. If you issue an ALTER SYSTEM statement to perform one of the preceding actions but nothing happens, verify that the log daemon is running.*

For the full syntax of the ALTER SYSTEM statement, refer the *SQL Reference Guide*.

Starting Logging

The ALTER SYSTEM START LOGGING statement starts logging. The log daemon performs the following actions:

- Creates a log file
- Begins accepting log request messages from database server processes and writes the corresponding log records to the file

If logging is already running, this statement is ignored.

Stopping Logging

The ALTER SYSTEM STOP LOGGING statement stops logging and closes the active log file. The log daemon continues running, and you can restart logging at any time. If logging is already stopped, this statement has no effect.

Switching Log Files

The ALTER SYSTEM SWITCH LOGGING FILE statement closes the active log file, creates a new active log file, and resumes logging to this new file. If logging is stopped, this statement has no effect.

Changing Log Filter Levels

The ALTER SYSTEM CHANGE LOGGING LEVEL statement changes the current log severity level for a selected log category. You can change any of the log categories to any severity level. This change takes effect immediately.

Terminating the Log Daemon

The ALTER SYSTEM TERMINATE LOGGING DAEMON statement terminates the log daemon process (**rbwlogd**). The log daemon closes and saves all active files (both log and account files) before shutting down. For information on account files and the accounting feature, refer to "Accounting" on page 8-34.

Query Logging

SQL statements for queries are logged through the USAGE ROUTINE event of the log file. To enable query logging, use the following statement:

```
RISQL> alter system change logging level usage routine;
```

To enable query logging, the log daemon must be running. When you enable logging, all queries that the database server processes are written to the log file. Depending on how active your system is, your log files might grow rapidly in size. If you enable query logging, provide ample disk space for your log files.

For the complete syntax of the ALTER SYSTEM statement, refer to the *SQL Reference Guide*.

Controlling Advisor Logging

There are separate log files for the Vista Advisor. This section describes the commands that control Advisor logging. For detailed information about using the Advisor, refer to the *Informix Vista User's Guide*.



Tip: *The Advisor logs cannot be read with the* ***rbwlogview*** *utility on UNIX or the* ***logdview*** *utility on Windows. The Advisor log files are read when you query the Advisor system tables.*

Advisor Log Files

The log daemon writes log records to the active Advisor log file. When this file exceeds the size specified by the ADMIN ADVISOR_LOG_MAXSIZE configuration parameter, the log daemon closes the file and creates a new active file. The log daemon also closes the active file when you stop the logging process and creates a new active file at logging startup. In this manner, a sequence of log files accumulates over time.

The active log file and the saved log files have the following naming conventions.

Operating System	Command
UNIX	rbwadvlog.<*daemon_name*>.active rbwadvlog.<*daemon_name*>.<*datetime_stamp*>
Windows	rbwadvlog.<*service_name*>.active rbwadvlog.<*service_name*>.<*datetime_stamp*>

The ***<datetime_stamp>*** suffix on saved filenames indicates the date and time at which the log daemon closed the file.

The default location for these files is the **$RB_CONFIG/logs** directory on UNIX and the **%RB_CONFIG%\logs** directory on Windows. To specify a different location, set the ADMIN ADVISOR_LOG_DIRECTORY configuration parameter. All log files are owned by the **redbrick** account.

Old log files are not removed automatically. The database administrator must provide a script to periodically remove these files or remove them manually.



Warning: *If you do not remove old log files, these files accumulate over time and can potentially consume all the free disk space. In addition, if you do not specify a value for ADMIN LOG_MAXSIZE, the log daemon writes to a single log file that grows until limited by available disk space.*

What the Advisor Logs

The Advisor logs the following information.

Information	Description
Time stamp	Indicates the date and time the message was logged.
Database name	Specifies the name of the database being used.
Base table identification	Integer that identifies the base table that was used to create the precomputed view.
View identification used to answer a query	Integer that identifies a precomputed view that was used to answer a query.

(1 of 2)

Information	Description
Rollup information	Integer that indicates the number of times a view was referenced to answer queries asking for a subset of the grouping columns of the view or asking for an attribute of a dimension with less granularity.
Elapsed time for the query and each aggregate block within the query	Integer that indicates the total amount of time spent executing the aggregate parts of a query.
SQL text for the aggregate block	Represents the definition of the candidate view.

(2 of 2)

Starting and Stopping the Advisor Log

Use the ADMIN ADVISOR_LOGGING parameter or the ALTER SYSTEM statement to start and stop Advisor logging.

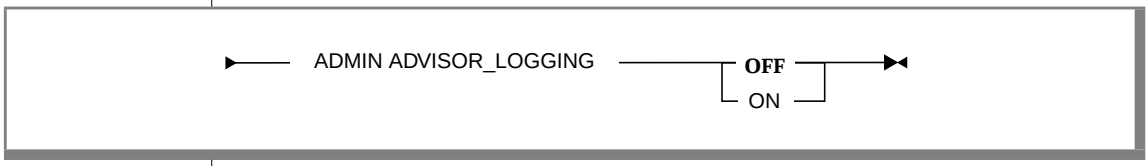
ADMIN ADVISOR_LOGGING

The ADMIN ADVISOR_LOGGING **rbw.config** file parameter determines the startup state of the Advisor log. When this parameter is set to ON, a log file is created when the log daemon starts. When this parameter is set to OFF, no log file is created, and data is not logged. The default setting is OFF.

If ADMIN ADVISOR_LOGGING is set to ON to create the log files *and* if OPTION ADVISOR_LOGGING is set to ON, the log records are captured when precomputed views are used and when candidate views are generated.

Syntax

The following syntax diagram shows how to set the ADMIN ADVISOR_LOGGING parameter.



ON / OFF — Specifies whether the Advisor log files are created at system startup. When this parameter is set to ON, a log file is created when the log daemon starts. When this parameter is set to OFF, no log file is created, and data is not logged.

ALTER SYSTEM

ALTER SYSTEM operations control database activity and various administrative actions. The two ALTER SYSTEM statements that control logging activities of the Advisor are the ADVISOR_LOGGING statement and ALTER SYSTEM SWITCH ADVISOR_LOG FILE statement. For more information, see the *SQL Reference Guide*.

ADMIN ADVISOR_LOG_DIRECTORY

The ADMIN ADVISOR_LOG_DIRECTORY **rbw.config** file parameter specifies the directory that stores the log files.

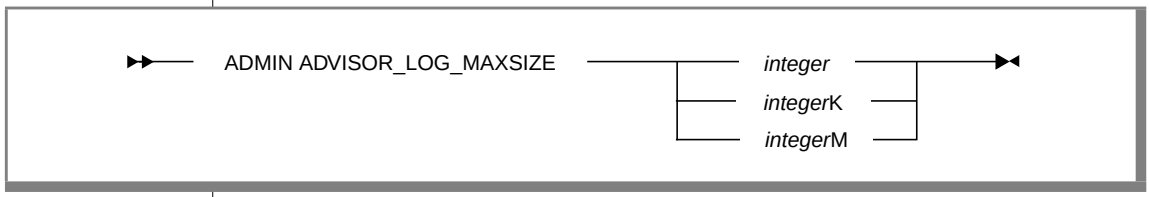
The following syntax diagram shows how to set the parameter ADMIN ADVISOR_ LOG_DIRECTORY.



The *pathname* variable specifies the directory in which log files are created. The *pathname* must specify an existing directory. It can be a relative or absolute pathname. Relative pathnames are relative to the directory specified by the **RB_CONFIG** environment variable. If the configuration parameter is not specified, the default logging directory is **$RB_CONFIG/logs** on UNIX or **%**RB_CONFIG**%\logs** on Windows.

ADMIN ADVISOR_LOG_MAXSIZE

The ADMIN ADVISOR_LOG_MAXSIZE parameter specifies the maximum advisor log file size. The syntax for setting this parameter is as follows.



integer, *integer*K, *integer*M — When a log file exceeds the size specified by this parameter, the log daemon closes the file and creates a new active file in the same directory. The units of the integer value are interpreted as follows:

- Bytes if neither K nor M is specified
- Kilobytes (1024 bytes) if K is specified
- Megabytes (1,048,576 bytes) if M is specified

If you specify K or M, this suffix must immediately follow the numeric value with no spaces. The minimum value for this parameter is 10 kilobytes (10,240 bytes).



Warning: *If you do not set this configuration parameter, or if you set this parameter to zero or a negative number, no maximum size is imposed on advisor log files. In this case, log files can continue to grow, limited only by available disk space in the log directory.*

Accounting

It is often useful to have a means of calculating the database workload that individual users generate. For example, you might want to implement a charge-back accounting system to charge users for their database use. The accounting feature described in this chapter provides a record of the database workload that each user generates.

The accounting feature generates records for database server operations that comprise the basic database workload and stores these records on disk. Accounting records are generated when a database server process (**rbwsvr**) or a TMU process (**rb_tmu**) completes any of the following operations:

- An individual DML operation
- A query
- A load operation

The accounting feature can run in two separate modes, *job* accounting and *workload* accounting. These modes differ in the level of workload detail captured.

Job accounting generates records that contain a summary of the resources used for a given operation. This summary includes statistics such as CPU time and elapsed time. Job accounting is intended for calculating the work that database users generate as a basis for cost accounting and charge-back systems.

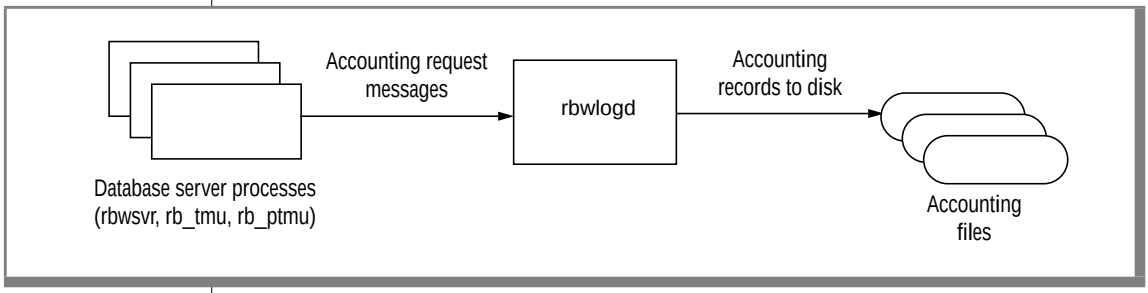
Workload accounting generates records that contain the same information as job accounting but include some additional detail. Workload accounting is intended primarily for system analysis by customer support.

You cannot display accounting records directly. Your site must have a program to read the accounting records and generate the appropriate data (for example, user charges) based on those records. The directory **redbrick_dir/util/readacct** on UNIX contains a sample program for accounting record processing. For more information, refer to the **README** file in this directory.

Accounting Process

Accounting is performed by the log daemon (**rbwlogd**). The process that generates accounting records is basically the same as the process that generates event log records. The **rbwsvr**, **rb_tmu**, or **rb_ptmu** processes send accounting request messages to the log daemon after performing basic workload operations. When a process generates an accounting request message, it does not wait for a response. To minimize performance impact, all communication between the generating process and the log daemon is one-way: from the process to the log daemon.

Figure 8-7
Accounting Process on UNIX



When the log daemon receives an accounting request message from an **rbwsvr**, **rb_tmu**, or **rb_ptmu** process, it adds a time stamp and writes the information contained in that message to a log file as a self-describing, variable-length record.

Format of Accounting Records

The accounting records are written in a Self-Describing Data Stream (SDDS) format. In this format, each record is variable length and contains the following elements:

1. A binary integer indicating the number of data bytes in the record
2. A binary type field indicating whether the record is a *job accounting* record or a *workload accounting* record
3. A series of encoded values each with a tag, type, and length header

   These values represent the actual accounting data.

The following table identifies the data type of each attribute (or *LA* parameter) in the accounting data and indicates its use.

Parameter	Data Type	Description
LA_USER_NAME	SDDT_CHAR	Database username
LA_SERVER_NAME	SDDT_INT4	Warehouse server name (the value set for the RB_HOST environment variable)
LA_DB_NAME	SDDT_CHAR	Logical name of the database
LA_ELAPSED_TIME	SDDT_INT4	Elapsed wall-clock time in 1/100ths of a second
LA_USRTIME	SDDT_INT4	User CPU time in 1/100ths of a second
LA_SYSTIME	SDDT_INT4	System CPU time in 1/100ths of a second
LA_IOREAD	SDDT_INT4	Number of logical reads
LA_IOWRITE	SDDT_INT4	Number of logical writes
LA_COMMAND_VERB	SDDT_CHAR	First one or two tokens (keywords) of the command
LA_MEMORY_USED	SDDT_INT4	Amount of memory used by the command (in kilobytes)
LA_SPILL_COUNT	SDDT_INT4	Number of spill files used in the operation
LA_CHILD_COUNT	SDDT_INT4	Number of child processes spawned
LA_SEL_ROWCNT	SDDT_INT4	Row count for a SELECT statement
LA_TMU_ROWCNT	SDDT_INT4	Row count for a TMU LOAD operation

(1 of 2)

Parameter	Data Type	Description
LA_SOURCEID	SDDT_CHAR	ID of the source component that ran statement
LA_PID	SDDT_CHAR	Process or thread ID of the component that ran the statement
LA_SQL_TEXT	SDDT_CHAR	First 256 characters of the SQL statement

(2 of 2)

The parameters described in this table are specific to the accounting file (**rbwacct**). They do not appear in the log file (**rbwlog**).

For online information about the accounting record format, refer to the **README** file and the sample code for processing accounting records, located in the ***redbrick_dir*/util/readacct** directory.

Accounting Files

The log daemon writes accounting records to the active accounting file. When this file exceeds the size specified by the ADMIN ACCT_MAXSIZE configuration parameter, the log daemon closes this file and creates a new active file. The log daemon also closes the active file when you stop the accounting process and creates a new active file at accounting startup. In this manner, a sequence of accounting files accumulates over time. The contents of the files in this sequence can be concatenated before processing because none of the files contains any header or trailer information.

The active accounting file and the saved accounting files have the following naming conventions.

Operating System	Command
UNIX	rbwacct.<*daemon_name*>.active rbwacct.<*daemon_name*>.<*datetime_stamp*>
Windows	rbwacct.<*service_name*>.active rbwacct.<*service_name*>.<*datetime_stamp*>

The ***<datetime_stamp>*** suffix on saved filenames indicates the date and time when the log daemon closed the file.

The default location for these files is the **$RB_CONFIG/logs** directory on UNIX and the **%RB_CONFIG%\logs** directory on Windows. To specify a different location, set the ADMIN ACCT_DIRECTORY configuration parameter. The **redbrick** account owns all accounting files.

Old accounting files are not removed automatically. The database administrator must provide a script to periodically remove these files or remove them manually.



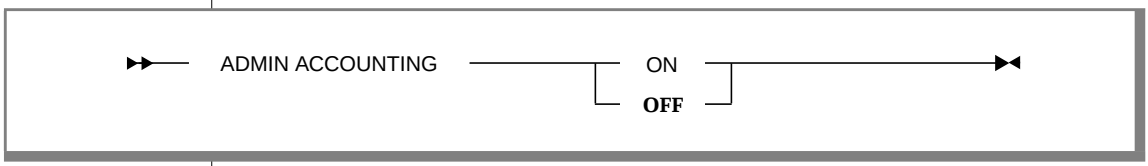
Warning: *If you do not remove old accounting files, these files accumulate over time and potentially consume all the free disk space. In addition, if you do not specify a value for ADMIN ACCT_MAXSIZE, the log daemon writes to a single accounting file that grows until limited by available disk space.*

Configuring Accounting

To configure the accounting subsystem, set the relevant configuration parameters in the **rbw.config** file.

Setting the Startup State

The ADMIN ACCOUNTING parameter determines the startup state of the accounting function. The syntax for setting this parameter is as follows.



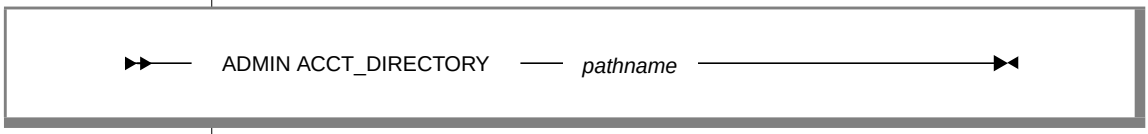
If this parameter is set to ON, the log daemon creates an accounting file when it starts and begins capturing accounting records. If this parameter is set to OFF, the log daemon does not create an accounting file or perform any accounting operations when it starts.



Tip: *When the log daemon is running, you can start or stop accounting via the ALTER SYSTEM START ACCOUNTING and ALTER SYSTEM STOP ACCOUNTING statements.*

Specifying the Location of Accounting Files

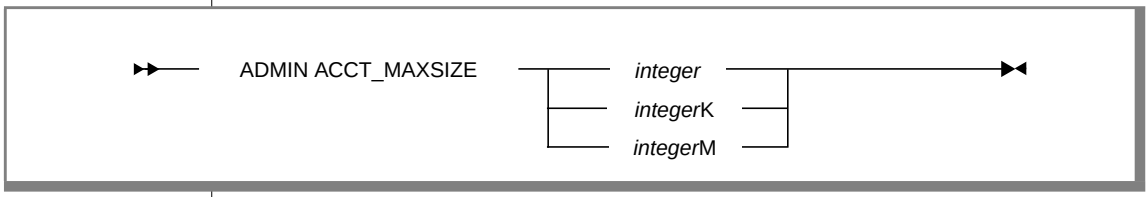
The ADMIN ACCT_DIRECTORY parameter specifies the directory in which the log daemon creates accounting files. The syntax for setting this parameter is as follows.



The *pathname* value can be an absolute pathname or a relative pathname. Relative pathnames are interpreted relative to the directory that the **RB_CONFIG** environment variable specifies. If you do not set the ADMIN ACCT_DIRECTORY parameter, the default accounting directory is **$RB_CONFIG/logs** on UNIX and **%RB_CONFIG%\logs** on Windows.

Specifying the Maximum Accounting File Size

The ADMIN ACCT_MAXSIZE parameter specifies the maximum accounting file size. The syntax for setting this parameter is as follows.



When an accounting file exceeds the size that this parameter specifies, the log daemon closes the file and creates a new active file in the same directory. The units of the integer value are interpreted as follows:

- Bytes if neither K nor M is specified
- Kilobytes (1,024 bytes) if K is specified
- Megabytes (1,048,576 bytes) if M is specified

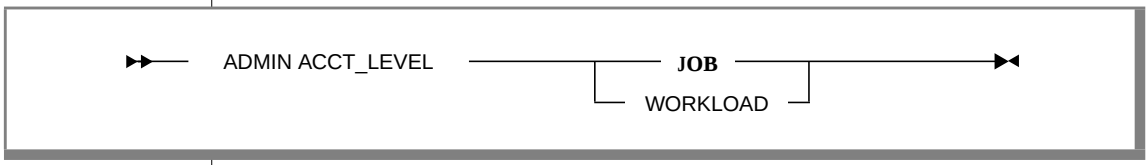
If you specify K or M, this suffix must immediately follow the numeric value with no spaces. The minimum value for this parameter is 10 kilobytes (10,240 bytes).



Warning: *If you do not set this configuration parameter, or if you set this parameter to zero or a negative number, no maximum size is imposed on accounting files. In this case, accounting files can continue to grow, limited only by available disk space in the accounting directory.*

Setting the Accounting Mode

The ADMIN ACCT_LEVEL parameter sets the accounting mode: JOB or WORKLOAD. The syntax for setting this parameter is as follows.



The level of detail captured in the accounting files depends on whether the database server is running job accounting or workload accounting. Job accounting includes basic resource utilization and is the default. Workload accounting includes additional information about each recorded event. Workload accounting has the potential to quickly produce large accounting files. Change the accounting detail level during database operation use with the ALTER SYSTEM CHANGE ACCOUNTING LEVEL statement.

Controlling Accounting

To control accounting operations while the database is running, use the ALTER SYSTEM statement. This statement has options for performing the following operations:

- Starting accounting
- Stopping accounting
- Switching accounting files
- Changing accounting mode

For the complete syntax of the ALTER SYSTEM statement, refer to the *SQL Reference Guide*.

Starting Accounting

The ALTER SYSTEM START ACCOUNTING statement starts accounting. The log daemon:

- Creates an accounting file
- Begins accepting accounting request messages from database processes and writes the corresponding accounting records to the file

If accounting is already running, this statement is ignored.

Stopping Accounting

The ALTER SYSTEM STOP ACCOUNTING statement stops accounting and closes the active accounting file. The log daemon continues running, and you can restart accounting at any time. If accounting is already stopped, this statement has no effect.

Switching Accounting Files

The ALTER SYSTEM SWITCH ACCOUNTING FILE statement closes the active accounting file, creates a new active accounting file, and resumes accounting, writing accounting records to this new file. If accounting is stopped, this statement has no effect.

Changing Accounting Mode

The ALTER SYSTEM CHANGE ACCOUNTING LEVEL statement changes the accounting mode from JOB to WORKLOAD or vice versa. This change takes place immediately.

Chapter

9

Maintaining a Data Warehouse

Locking Tables and Databases 9-6
Manual Table or Database Locks 9-6
Schema Locking 9-7
Types of Table Locks 9-8
Locking and Segments 9-10
Determining When to Lock a Table or Database 9-10
Specifying Wait Behavior for Database Server and TMU Locks . . 9-11
No-Wait Behavior 9-11
Deadlocks 9-12
Setting Isolation Level for Versioned Transactions 9-12

Obtaining Information on Tables and Indexes 9-14
Checking and Repairing Table Structure 9-14
CHECK TABLE Segment Statistics 9-14

Monitoring Growth of Tables and Indexes 9-18
STAR Indexes 9-19
MAXSIZE Column 9-20
USED Column 9-21
TOTALFREE Column 9-22
Pseudocolumns. 9-23
Freespace Field 9-24

Adding Space to a Segment 9-26

Altering Segments 9-29
Using ALTER SEGMENT 9-29
Ensuring No Users Are Active 9-30
Attaching and Detaching Segments 9-31
Moving Entire Segments 9-31

Specifying a Segmenting Column 9-32
Specifying a Range. 9-32
Taking a Segment Offline or Online 9-32
Clearing a Segment 9-33
Renaming a Segment 9-33
Changing PSU Sizes 9-33
Changing PSU Location 9-34
Verifying a Segment 9-34
Forcing a Segment into an Intact State 9-35

Recovering a Damaged Segment 9-35

Managing Optical Storage 9-37
Assigning Optical Storage 9-38
Specifying Access Behavior for Optical Segments 9-39
Specifying Index Selection with Optical Segments 9-40

Altering Tables 9-40
Adding and Dropping Columns 9-41
Changing a Column Name 9-42
Changing the Default Value for a Column. 9-42
Changing the MAXSEGMENTS and MAXROWS PER SEGMENTS Values 9-42
Changing the Way Referential Integrity Is Maintained 9-43
Changing the Data Type for a Column 9-43
Adding and Dropping Foreign Keys 9-44
Changing the Fill Factor for a VARCHAR Column. 9-45
Changing the Start and Step Values of a SERIAL Column 9-45
Recovering from an Interrupted ALTER TABLE Operation 9-46
Recovering the Table. 9-46
Interruptions: Causes and Prevention 9-47

Copying or Moving a Database. 9-48
Full Versus Relative Pathnames 9-48
Copy7ing a Database That Contains Only Relative Pathnames . . 9-50
Copying a Database That Contains Full Pathnames 9-50
Moving a Database That Contains Only Relative Pathnames . . . 9-51
Moving a Database That Contains Full Pathnames. 9-52

Modifying the Configuration File 9-52

Monitoring and Controlling a Database Server 9-59
Monitoring and Controlling a Database Server on UNIX 9-59
Daemon Processes 9-60
Findserver Utility 9-61
Log Files 9-62
Monitoring and Controlling a Database Server on Windows 9-62

Determining Version Information 9-64

Deleting Database Objects and Databases. 9-64
Dropping Database Objects 9-65
Indexes 9-65
Hierarchies 9-66
Macros 9-66
Roles. 9-66
Segments 9-67
Synonyms 9-67
Tables 9-67
Views 9-68
Deleting a Database 9-68

In This Chapter

This chapter discusses ongoing tasks involved in maintaining a data warehouse to meet users' needs or reflect changes to a database. For a database that does not change except when the entire database is loaded with new data, maintenance tasks are minimal. You need only ensure that enough space is available in the default and named segments to accommodate the current batch of input data. Any needed restoration of the database can be done from the original input data files.

For databases that are modified by incremental load operations or INSERT, UPDATE, or DELETE statements, maintenance includes accommodating growth in the database, as well as adjusting to changes in users' needs or the warehouse environment. For databases that change, backing up the data regularly is an important part of maintenance. For information about backup operations, refer to your system utilities documentation.

This chapter includes the following sections:

- Locking Tables and Databases
- Obtaining Information on Tables and Indexes
- Monitoring Growth of Tables and Indexes
- Adding Space to a Segment
- Altering Segments
- Recovering a Damaged Segment
- Managing Optical Storage
- Altering Tables
- Copying or Moving a Database
- Modifying the Configuration File

- Monitoring and Controlling a Database Server
- Determining Version Information
- Deleting Database Objects and Databases

Locking Tables and Databases

To preserve consistency within a database, operations that modify data must be allowed to complete without interruption, blocking other read or write operations until the modification is complete. Locking is performed automatically by operations that require it for database consistency. For instance, the TMU automatically locks the tables and database as needed for its operations. Tables that are to be modified are locked with a write lock, and tables that are used for checking referential integrity are locked with a read lock. Other operations, such as cleaning the version log, require a manual lock.

Manual Table or Database Locks

Manual locks allow for a sequence of operations on a table or database.

A user can manually lock only one object (table or database) at a time with the LOCK statement. The first object must be unlocked before a second one can be locked. However, the system can implicitly lock as many objects as necessary to process a query or command.

The LOCK DATABASE statement supports two database lock options:

- WRITE locks the database to prevent any other access to it. WRITE is the default option for the LOCK DATABASE statement.
- READ locks the database to prevent updates but allows concurrent queries, which use read locks. Read locks do not affect vacuum cleaning. Read locks are useful when backing up versioned databases. For more information on backing up databases, see "Backup and Recovery" on page 6-24.

For a complete description of the LOCK statement, refer to the *SQL Reference Guide*.

Schema Locking

Locking is affected by the READER_PRIORITY option. By default, the READER_PRIORITY option is set to ON, which gives priority to read locks at the expense of write locks. For example, if a user has a read lock on a table and another user requests a read lock on the same table, that request is granted immediately, even if a third user is requesting a write lock on the table.

For applications that require activities such as DDL operations and loads while the database is online, setting OPTION READER_PRIORITY OFF in the **rbw.config** file reduces contention in two ways:

- Operations get locks in the order for which they are applied, with no priority given to read locks for queries.
- The system table lock held for queries is released sooner, effectively reducing the wait for any operation that needs to lock and update the system catalog.

Conversely, this approach can delay the locking of tables required for query execution. Unless you routinely encounter problems with delayed DDL operations that must be performed while the database is online, Informix recommends that you leave OPTION READER_PRIORITY set to ON.

When this option is set to ON, the default, queries requesting read locks on tables have priority over operations that need to write to the database. When queries are executed, they must acquire a database-level lock on the system tables. This lock is not released until the individual tables are locked. This behavior is appropriate for most decision-support applications.

However, heavy query loads can cause any concurrent DDL operations to be significantly delayed because each new request for a database read lock is given higher priority than requests for database write locks. In this case, setting READER_PRIORITY to OFF disables query-priority locking in general and enables *schema locks* to be acquired at the table level, instead of relying on a database-level lock on the system tables. As soon as the schema locks are acquired, the database lock can be released.

When READER_PRIORITY is set to OFF, the **DST_LOCKS** table and the **showlocks** utility reflect two locks: RS (read schema) and WS (write schema).

Types of Table Locks

To check what types of locks are on tables, query the **TYPE** column of the **DST_LOCKS** table.

For databases without versioning, Red Brick Decision Server provides a read-only (RO) lock and a write-blocking (WB) lock.

For databases with versioning, Red Brick Decision Server provides additional locks. Read locks for versioning lock a table for read access only, allowing multiple readers but restricting write actions. This access ensures that a transaction reads a consistent view of a table. Write locks for versioning allow other users various levels of read access to the table for the duration of the lock.

Different types of read and write locks allow varying levels of concurrency in the database, ranging from disallowing any concurrent transactions (blocking) to allowing a write transaction on a table while read transactions are taking place on versions of the same table or a table with a primary key / foreign key relationship.

For greater concurrency, schema locks can have table-level locks instead of database-level locks.

The following table provides definitions and descriptions of the types of locks.

Lock Type	Definition	Used for This Type of Transaction
RO	Read-only	Read transaction that allows any other read or write transactions except a blocking transaction on the table.
RK	Read-key	Read transaction that does not allow other transactions to change primary key values in the table. Versioned databases only.
RD	Read-data	Read transaction that does not allow any write transactions to the table. Versioned databases only.

(1 of 2)

Lock Type	Definition	Used for This Type of Transaction
WD	Write-data	Write transaction that does not modify existing primary key values in the database (for example, INSERT or UPDATE of nonkey columns). Versioned databases only.
WK	Write-key	Write transaction that modifies existing primary key values in the database (for example, DELETE or UPDATE of key columns). Versioned databases only.
WB	Write-blocking	Write transaction that does not allow any other read or write transactions on the table.
RS	Read-schema	Only used when READER_PRIORITY is set to OFF. See "Schema Locking" on page 9-7.
WS	Write-schema	Only used when READER_PRIORITY is set to OFF. See "Schema Locking" on page 9-7.

(2 of 2)

The following table shows how the various types of locks interact. Read-schema and write-schema are not included as they do not interact with other locks.

The shaded area indicates blocking transactions (transactions without versioning) for the lock combinations. The unshaded area indicates concurrent transactions (versioned transactions).

	RO	RK	RD	WD	WK	WB
RO	C	C	C	C	C	B
RK	C	C	C	C	B	B
RD	C	C	C	B	B	B
WD	C	C	B	B	B	B
WK	C	B	B	B	B	B
WB	B	B	B	B	B	B

Locking and Segments

Individual segments cannot be explicitly locked. If a table is locked, access to all online segments therein is also locked. Operations permitted on offline segments automatically secure the necessary locks. For example, an offline load operation automatically write-locks that segment to prevent two simultaneous load or restore operations, and it automatically read-locks the owning table to prevent the segment from being dropped or altered during the offline operation.

Determining When to Lock a Table or Database

You can choose to set a read lock on a database, or a write lock on a table or database, according to your requirements.

You should set a read lock on the database when you back it up. A read lock allows queries but prevents updates.

You might choose to set a write lock on a table in the following cases:

- To perform consecutive modifications to the data in a table

  Lock the table to prevent access by other users until all the modifications are complete.

- Before beginning a delete operation to ensure the access needed to maintain referential integrity of affected tables (with LOCK...FOR DELETE on those tables)

  Although all required locking is performed automatically, the DELETE operations can complete sooner if you manually lock the tables to prevent access by other users between DELETE operations. For information on how the ON DELETE clause works and which tables are locked, refer to "Maintaining Referential Integrity with ON DELETE" on page 5-15.

Set a write lock on a database when you are performing a restore operation or any operation that affects the entire database. Database locks are applied with the LOCK DATABASE statement. These locks prevent the entire database, all tables including system tables, from being read or modified by anyone else until the UNLOCK DATABASE statement is issued. If this lock is applied to a database, the system tables are locked so that no new operations that access these tables can start until the database is unlocked.

You might also choose to set a write lock the database in the following cases:

- You are executing a consecutive series of ALTER TABLE statements and want to prevent any intervening operations by other users.
- You are performing operations that affect the system tables, such as ALTER SEGMENT operations.

Specifying Wait Behavior for Database Server and TMU Locks

When a database server or TMU process encounters a read-locked table or database, the default behavior is that the process waits until it can obtain the lock. The way in which the process waits depends on the setting of the READER_PRIORITY option. The behavior can also be affected by the setting of the NO WAIT option. If the wait could cause a deadlock, the operation fails unless the ALLOW_POSSIBLE_DEADLOCK parameter is set.

No-Wait Behavior

You can change the locking behavior so that instead of the process waiting, a message indicates that the database server or TMU operation failed because a table or database was locked.



Important: *Informix does not recommend the NO WAIT option for versioned databases.*

To change the wait behavior for TMU activity, include a TMU SET LOCK NO WAIT command in a TMU control file. For more information, refer to the *Table Management Utility Reference Guide*.

To change the wait behavior for database server statements such as ALTER SEGMENT, INSERT, UPDATE, or DELETE, specify NO WAIT by any of the following means:

- Interactively for the current session with a SET LOCK statement
- For all of a user's database server sessions with a SET command in that user's **.rbwrc** file
- For all database server sessions in the database server **.rbwrc** file

Deadlocks

If waiting for existing locks to be released might result in a deadlock, the database server denies the lock request and immediately returns control to the lock requestor. If you prefer to risk occasional deadlocks in exchange for the WAIT option to always wait, you can include the following line in the **rbw.config** file:

```
OPTION ALLOW_POSSIBLE_DEADLOCK ON
```

If you set this option ON and deadlocks occur, you must then use a system command to kill all deadlocked processes. The default value of this parameter is OFF.

Deadlocks occur only when the LOCK TABLE or LOCK DATABASE statement is used. The automatic locking operations of the database server or TMU do not cause deadlocks. Instead, the database server returns an error message before the deadlock occurs.

Setting Isolation Level for Versioned Transactions

For versioned SQL transactions, two user-controlled *isolation levels* are available: Serializable and Repeatable Read. The isolation level is the level of concurrency allowed for the duration of the transaction. Depending on the level being used, different read locks are used on the tables to allow different levels of access to those tables. For information on lock types, refer to "Types of Table Locks" on page 9-8.

The Serializable mode is the more restrictive. Other versioned transactions cannot read a table locked by another transaction to modify another table. This mode ensures that the new versioned transaction reads the latest version of the table before using it to modify another table.

The Repeatable Read mode is less restrictive. Versioned transactions can read an older version of a table that is locked by another transaction to modify another table. This mode allows a new versioned transaction to proceed with an older version of the table, allowing the new transaction to execute sooner but with no guarantee that it is using the latest version of one table to modify another table.

The isolation level is controlled through the SET TRANSACTION ISOLATION LEVEL statement and the OPTION TRANSACTION_ISOLATION_LEVEL parameter in the **rbw.config** file. The default level is Serializable. For the syntax of this statement, refer to the *SQL Reference Guide*.

Example

To illustrate the difference between the isolation levels, consider an application with a 100,000,000-row customer table. Assume that the table has a versioned transaction (a TMU LOAD operation) that just started and is adding new customers whose last names begin with the letter *A*. You know that this transaction takes about 15 minutes to complete.

Your manager suddenly calls and asks you to add all the customers whose last name starts with the letter *S* to the **Preferred Customer** table. The **Preferred Customer** table is in the same database but is not related to the **Customer** table with any primary key / foreign key relationships. You need to complete the operation in 10 minutes. The SQL INSERT statement for this operation is as follows:

```
insert into preferred_customer
   select last_name, first_name, id, address from customer
   where last_name like 'S%'
   group by last_name, first_name, id, address
   order by last_name, first_name;
```

If you run this transaction in Serializable mode, the LOAD operation must first complete. The transaction must wait because it is using data from a table that is currently being modified to modify another table. But you know that you are not going to use any of the data that is currently being modified. You are interested in customers with last names beginning with *S*, and only customers with last names beginning with *A* are being changed.

You run the transaction in Repeatable Read mode by issuing the following statement before executing the INSERT statement:

```
set transaction isolation level repeatable read;
```

Then you run the query, and it executes without waiting.

In this example, if the LOAD operation added new customers whose last names begin with the letter *S* instead of the letter *A*, and if you run the INSERT operation in Repeatable Read mode, you might not get the results you expect. You do not see the latest changes made with this INSERT operation.

Obtaining Information on Tables and Indexes

You can use the CHECK TABLE statement to:

- Check for damage to the table storage data structure
- Repair damage to the table storage data structure
- Obtain segment statistics for tables, including row counts

You use the CHECK INDEX statement to check an index for corruption and obtain configuration and size information.

Checking and Repairing Table Structure

When you specify the FIX option, the CHECK TABLE statement marks as damaged any damaged segment that it finds. After you repair the damage, use the ALTER SEGMENT...VERIFY statement to verify that the segment has been repaired and to mark it as undamaged. Complete this step before you use the CHECK TABLE statement on the table again.



Warning: *Informix recommends that before you use the CHECK TABLE statement with the FIX option, you make sure that a current backup of your database exists and contact Informix Customer Support.*

CHECK TABLE Segment Statistics

When you specify the VERBOSE option, the CHECK TABLE statement generates the following segment statistics in the output file. For a description of the segment statistics that CHECK TABLE produces, refer to "Using CHECK TABLE with the VERBOSE Option" on page 10-35.

CHECK TABLE Segment Statistics Field	Description
active rows	Number of rows that reside in the segment.
deleted rows	Number of rows deleted from the segment.
total blocks	Number of 8-kilobyte blocks in the segment.

(1 of 3)

CHECK TABLE Segment Statistics Field	Description
free blocks	Number of blocks that currently do not contain data rows.
storage reclen	Length (in bytes) that the database server expects each record to be based on the fill factor of VARCHAR columns. To calculate this value, see the typical row size formula in "Example 1: Effect of the VARCHAR Fill Factor on Number of Rows Per Block" on page 10-31.
average reclen	Actual average size (in bytes) of records (rows) in the segment. The average row size differs from the **storage reclen** value when the row contains one or more VARCHAR columns.
indirect rows	Number of rows that reside on a different page than where they originally were loaded or inserted. Indirect rows occur when not enough space exists on the page to store a longer row that resulted from an update of a VARCHAR column.
max reclen	Length (in bytes) of the row when all VARCHAR columns contain the longest possible string.
longest rec	Actual longest row length (in bytes) that exists in the segment.
min reclen	Length (in bytes) of the row when all VARCHAR columns are null or empty.
shortest rec	Actual shortest row length (in bytes) that exists in the segment.
freespace	Number of bytes of unused space in the allocated blocks in a segment. See "Example: Freespace Segment Statistics in CHECK TABLE Output" on page 9-25.
rows/block	Number of row numbers the database server allocates per block based on the fill factor of VARCHAR columns. For the formula to calculate this value, see "How the Database Server Uses the VARCHAR Fill Factor" on page 10-29.

(2 of 3)

CHECK TABLE Segment Statistics Field	Description
unused rowids	Number of row numbers currently not used by actual rows
unusable freespace	Number of bytes of free space in blocks where all assigned row numbers are used and the amount of free space exceeds the size of a typical row. Nonzero values appear in this field when the fill factor is higher than the actual row size, as illustrated in "Example 2: VARCHAR Fill Factor Too High" on page 10-32.

(3 of 3)

Example: Segment Statistics in CHECK TABLE with VERBOSE Option

This example shows the CHECK TABLE statement with the VERBOSE option for the **Sales** table from the **Aroma** database.

```
RISQL> check table sales
> directory '/qa/local/sct-pubs/varchar-aroma/'
> verbose;

INFORMATION
      Table: 11  Segment: 21 is ok
      Table: 11  Segment: 22 is ok
      No inconsistencies were detected.
```

Because the **Sales** table consists of two segments, the CHECK TABLE statement produces two output files. The name of each output file for the previous CHECK TABLE statement starts with the table and segment numbers, as the following filename shows:

```
chk_11_21_rep.19990908.184731.25596
chk_11_22_rep.19990908.184731.25599
```

Figure 9-1 shows the contents of the output files that the CHECK TABLE statement with the VERBOSE option produces for the **Sales** table.

In this example, the CHECK TABLE output shows equal values for the following segment statistics because no VARCHAR columns exist in the table: **storage reclen**, **average reclen**, **max reclen**, **longest rec**, **min reclen**, and **shortest rec.**

Figure 9-1
Contents of Two Output Files from CHECK TABLE with VERBOSE Option

```
RISQL> !more chk_11_21_rep.19990923.135804.19354
------- Checking Table -------

Table name:   SALES
Table revision no: NULL
Segment name: DAILY_DATA1
------------------------------

Validating physical storage structure for segment...
Finished checking segment physical structure.
Validating row structure for segment...
Expecting:34577 active rows in segment
Scanning all active blocks. Total data blocks:131...
Segment statistics:
  active rows:34577, deleted rows:0, total blocks:132, free blocks:0
  rows/block:264, storage reclen:29, average reclen:29, indirect rows:0
  max reclen:29, longest rec:29, min reclen:29, shortest rec:29
  freespace:741, unused RowIDs:7, unusable freespace:0
Finished checking segment row structure.


RISQL> !more chk_11_22_rep.19990923.135804.19356
------- Checking Table -------

Table name:   SALES
Table revision no: NULL
Segment name: DAILY_DATA2
------------------------------

Validating physical storage structure for segment...
Finished checking segment physical structure.
Validating row structure for segment...
Expecting:35364 active rows in segment
Scanning all active blocks. Total data blocks:134...
Segment statistics:
  active rows:35364, deleted rows:0, total blocks:135, free blocks:0
  rows/block:264, storage reclen:29, average reclen:29, indirect rows:0
  max reclen:29, longest rec:29, min reclen:29, shortest rec:29
  freespace:908, unused RowIDs:12, unusable freespace:0
Finished checking segment row structure.
```

The value in the **rows/block** field is the same as the rows per block that the database server calculates with the following formula:

```
rows per block
         = floor((block size - overhead) / (row size + overhead))
         = floor((8192 - 4) / (29 + 2))
         = 264
         = CHECK TABLE rows/block:264
```

The value in the **freespace** field is the number of unused bytes in existing allocated blocks. For more information, refer to "Freespace Field" on page 9-24.

For a description of how to use the CHECK TABLE segment statistics to determine the effectiveness of the VARCHAR column fill factor, refer to "Using CHECK TABLE with the VERBOSE Option" on page 10-35.

Monitoring Growth of Tables and Indexes

If tables and their indexes grow in your database, you must accommodate this growth. To prevent the database from running out of space at an inconvenient time, monitor the growth and compare the actual growth with the space available, adding new segments, PSUs, or both as needed. Error messages are issued when a segment is full.

If you receive an out-of-space error because a segment ran out of space, you can either:

- Specify a larger value for the MAXSIZE value of the last PSU with the ALTER SEGMENT…CHANGE MAXSIZE option
- Add a new PSU to the segment with the ALTER SEGMENT…ADD STORAGE option



Important: *If no available file system contains additional space, you must make file system space available before you can add more data to the segment.*

To monitor the growth of a table or index, use the information in the **RBW_SEGMENTS** table (**TNAME** or **INAME**, **NPSUS**, and **TOTALFREE**, ROWCOUNT columns) and **RBW_STORAGE** table (**SEGNAME**, **MAXSIZE**, and **USED** columns). Default segments grow as needed but are limited by file system space.

STAR Indexes

When a STAR index is built, its size is based on the maximum number of rows in the referenced tables, which is calculated based on the values specified in the MAXROWS PER SEGMENT and MAXSEGMENTS parameters. If you change the values of these parameters to exceed the maximum number of rows used to build the STAR index, a message indicates that the STAR indexes based on that table might not be valid and might need to be either rebuilt with the REORG command or dropped and re-created. For information about STAR index growth, refer to "Considerations for Growing Tables" on page 4-48.

To ensure that a STAR index is built with sufficient space to accommodate the expected growth of the corresponding referenced tables, create each referenced table with MAXROWS PER SEGMENT and MAXSEGMENTS values that accurately reflect the expected size.

If an error message indicates that one or more STAR indexes are invalid, you must either perform a REORG operation on those indexes to rebuild the invalid STAR indexes or drop and re-create the STAR indexes to accommodate table growth. If space still remains in a default segment, you can use the REORG command to rebuild the index. If any segments containing the STAR indexes are full, you must use the ALTER SEGMENT statement to make space available or create and attach additional segments to the index before performing the REORG operation.

MAXSIZE Column

Each PSU is divided into 8-kilobyte (8192-byte) blocks, the minimum allocation unit for disk storage in Red Brick Decision Server. The **MAXSIZE** column in the **RBW_STORAGE** system table specifies the maximum size in kilobytes to which a specific PSU (file) in a segment is allowed to grow. The values in the **MAXSIZE** and **INITSIZE** columns do not necessarily match the **MAXSIZE** and **INITSIZE** numbers in the CREATE SEGMENT statement for the storage file for several reasons:

- The values in the **MAXSIZE** columns are always rounded up to the nearest 8 kilobytes.
- The first file always contains at least 2 blocks, or 16 kilobytes.
- **MAXSIZE** values are dynamically adjusted in certain cases where a file system runs out of space before a PSU in that file system reaches its **MAXSIZE** value.

Example

The following statement creates a segment containing two PSUs, **mkt1** and **mkt2**:

```
create segment mkt
      storage 'mkt1' maxsize 38 initsize 16 extendsize 8
      storage 'mkt2' maxsize 30;
```

The **RBW_STORAGE** table shows a maximum size of 40 kilobytes for **mkt1** and a maximum size of 32 kilobytes for **mkt2** (rounded up to the nearest multiple of 8 kilobytes).

The **MAXSIZE** column in the **RBW_STORAGE** table is an upper limit on the file size. If the file system on which the PSU resides becomes full before the PSU has grown to its maximum size and space is available in subsequent PSUs in the segment, the system dynamically reduces the **MAXSIZE** value of the PSU to its current size and starts using the next PSU with space available.

USED Column

The **USED** column in the **RBW_STORAGE** system table indicates how much of the PSU has been allocated so far—the largest amount of space the PSU has ever occupied. This number also provides the lower limit on the new **MAXSIZE** in the CHANGE MAXSIZE option for the ALTER SEGMENT statement.

The **USED** value is not necessarily the amount of space used in the PSU because some of the **USED** space might actually be on an internal free-storage list. To determine how much free space is available in the used blocks, use the CHECK TABLE statement with the VERBOSE option, as described in "Freespace Field" on page 9-24.

Example: MAXSIZE and USED Columns in RBW_STORAGE

The following example shows the values of the **MAXSIZE** and **USED** columns in the **RBW_STORAGE** system table for the **Sales** table from the **Aroma** database:

```
RISQL> select segname,segid,pseq,maxsize,initsize,extendsize,used
> from rbw_storage where segid = 21;

SEGNAME     SEGID PSEQ MAXSIZE INITSIZE EXTENDSIZE  USED
DAILY_DATA1    21    1    1024      104        104  1024
DAILY_DATA1    21    2    1024      104        104    32

RISQL> select segname, segid, maxsize, initsize, extendsize, used
> from rbw_storage where segid = 22;

SEGNAME     SEGID PSEQ MAXSIZE INITSIZE EXTENDSIZE  USED
DAILY_DATA2    22    1    1024      104        104  1024
DAILY_DATA     22    2    1024      104        104    56
RISQL>
```

The **USED** column shows that the first PSU in segments DAILY_DATA1 and DAILY_DATA2 have reached the MAXSIZE value of 1024 kilobytes. A second PSU was allocated to each segment.

TOTALFREE Column

The **TOTALFREE** column in the **RBW_SEGMENTS** system table contains the amount of free space available to the segment, whether the segment is associated with an index or a table. This value assumes that the file systems contain sufficient space to allow the segment to grow to its maximum size.

For tables, if no rows have been deleted from a table, the difference between **MAXSIZE** and **USED** space in **RBW_STORAGE** equals **TOTALFREE** space in **RBW_SEGMENTS** for the segments associated with that table. See "Example: TOTALFREE Column in RBW_SEGMENTS" on page 9-22.

Red Brick Decision Server reuses space by row. When a row is deleted from a table, the next row added to the table is stored in the location of the last deleted row. Therefore, after several rows residing in a given segment have been deleted from a table, that segment contains free space where those rows were stored. The value for **TOTALFREE** measures only the space that has not yet been used, not the space freed by deleting rows. If you have deleted large numbers of rows from your table, you might have more free space than the value of **TOTALFREE** indicates.

To see how much free space is available in the used blocks, use the CHECK TABLE statement with the VERBOSE option, as described in "Freespace Field" on page 9-24.

Example: TOTALFREE Column in RBW_SEGMENTS

The following query obtains the **TOTALFREE** column in the **RBW_SEGMENTS** system table for the **Sales** table from the **Aroma** database:

```
RISQL> select name, tname, npsus, ncols, id, totalfree
> from rbw_segments where tname = 'SALES';

NAME                TNAME   NPSUS   NCOLS   ID   TOTALFREE    INAME
DAILY_DATA1         SALES   2       1       21       992     NULL
DAILY_DATA2         SALES   2       1       22       968     NULL
DEFAULT_SEGMENT_23 SALES    1       1       23   2096304     SALES_STAR_IDX
```

"Example: MAXSIZE and USED Columns in RBW_STORAGE" on page 9-21 shows these values for the **Sales** table from the **Aroma** database. The following equation shows that no rows have been deleted:

```
UNALLOCATED SPACE = MAXSIZE - USED
                  = 1024 - 32
                  = 992 kilobytes
                  = TOTALFREE
```

Pseudocolumns

Every user table in a Red Brick Decision Server database has three *pseudocolumns*: **RBW_ROWNUM**, **RBW_SEGID**, and **RBW_SEGNAME**. A pseudocolumn does not take up any space in the system tables but instead selects bytes stored in the headers of blocks and displays this information in a column format. Do not include pseudocolumns in calculations of table size.

If you issue a query with pseudocolumns in the select list, they are written as columns in the query result. These columns should therefore be part of any storage space calculations you perform on the query results.

Pseudocolumn	Data Type	Description
RBW_ROWNUM	INTEGER	Contains the row number for each row in a segment, where the first row in the segment is the number 0. Each segment begins its count of rows with the number 0. Therefore, if you have multiple segments, you will have multiple rows where **RBW_ROWNUM** is equal to a particular value.
RBW_SEGID	SMALLINT	Contains a relative segment ID for a given row. A lower value indicates a segment that comes before one with a higher value. This value corresponds to the value in the **LOCAL_ID** column of the **RBW_SEGMENTS** system table. The values in the **RBW_SEGID** pseudocolumn are not necessarily consecutive. The relative order of the values allows you to determine the relative order of the segment.
RBW_SEGNAME	CHAR(129)	Contains the name of the segment where the data corresponding to a given row is stored.

Example

The following example shows the values of the **RBW_ROWNUM**, **RBW_SEGID**, and **RBW_SEGNAME** pseudocolumns for a query on the **Sales** table from the **Aroma** database:

```
RISQL> select rbw_rownum, substr(rbw_segname, 1, 20)
> as RBW_SEGNAME, rbw_segid, dollars
> from sales where rbw_rownum < 4;

RBW_ROWNUM  RBW_SEGNAME       RBW_SEGID    DOLLARS
         0  DAILY_DATA1               0     34.00
         1  DAILY_DATA1               0     60.75
         2  DAILY_DATA1               0    270.00
         3  DAILY_DATA1               0     36.00
         0  DAILY_DATA2               1    348.00
         1  DAILY_DATA2               1    123.25
         2  DAILY_DATA2               1    121.50
         3  DAILY_DATA2               1     56.00
RISQL>
```

In this example, two rows correspond to the values 0, 1, 2, and 3 in the **RBW_ROWNUM** pseudocolumn: one resides in the DAILY_DATA1 segment, and one resides in the DAILY_DATA2 segment.

Freespace Field

The **freespace** field in the output of the CHECK TABLE statement with the VERBOSE option displays the number of bytes in existing allocated blocks in the table that are not used. To obtain a more accurate value of unused space within the segment, add this **freespace** value to the **TOTALFREE** column in the **RBW_SEGMENTS** system table.

Example: Freespace Segment Statistics in CHECK TABLE Output

The following example shows the values of the **freespace** field in the CHECK TABLE output and of the **TOTALFREE** column in the **RBW_SEGMENTS** system table for the **Sales** table from the **Aroma** database:

```
Segment statistics:
  active rows:34577, deleted rows:0, total blocks:132, free blocks:0
  rows/block:264, storage reclen:29, average reclen:29, indirect rows:0
  max reclen:29, longest rec:29, min reclen:29, shortest rec:29
  freespace:741, unused RowIDs:7, unusable freespace:0

Segment statistics:
  active rows:35364, deleted rows:0, total blocks:135, free blocks:0
  rows/block:264, storage reclen:29, average reclen:29, indirect rows:0
  max reclen:29, longest rec:29, min reclen:29, shortest rec:29
  freespace:908, unused RowIDs:12, unusable freespace:0
```

In this example, the *freespace* field value is 741 bytes for the first segment. The following equations show that this *freespace* value is the unused bytes in the allocated space:

```
space allocated per block
   = ((storage reclen +overhead)* rows/block) + overhead
                          = ((29 + 2)* 264) + 4
                          = 8188 bytes / block

space allocated in segment 1
   = bytes allocated per block * number of blocks
   = 8188 * 131
   = 1072628 bytes allocated per segment

actual space used = active rows * (average reclen +overhead)
            = 34577 * (29 + 2)
            = 1071887 bytes used in segment 1

freespace = space allocated per segment - actual space used
   = 1072628 - 1071887
   = 741 bytes
```

"Example: TOTALFREE Column in RBW_SEGMENTS" on page 9-22 shows that the **TOTALFREE** value for the **Sales** table in the **Aroma** database is 992 kilobytes. Use the following equation to calculate the total unused space in the first segment of the **Sales** table:

```
total unused space = TOTALFREE + freespace
                   = 992 kilobytes + 741 bytes
                   = 992.72 kilobytes
```

Adding Space to a Segment

A segment runs out of space only when all available space in the segment is allocated, as indicated by a value of `0` for the corresponding **TOTALFREE** column in the **RBW_SEGMENTS** system table.

If a PSU runs out of space because the file system is full, the PSU MAXSIZE parameter is dynamically decreased to the current size of the file. If space is available in subsequent PSUs in other file systems, the operation continues.

PSUs are used sequentially in the order in which they were defined (by sequence number in the **RBW_STORAGE.PSEQ** column). The current PSU is the one being written to. More space can be added to a PSU only if the segment containing the PSU is unattached or if the PSU is the current PSU or a subsequent PSU. (In other words, you can never add space to a previous PSU once the next PSU is in use.)

When segments and PSUs are created, only the amount of storage specified by the INITSIZE value for each PSU is allocated immediately. The rest of the storage is not allocated until it is needed. Consequently, a file system can fill before a PSU reaches the limit specified by its MAXSIZE parameter. If this situation occurs, the database server automatically checks the subsequent PSUs for available space—either already allocated but unused INITSIZE space or space on another file system.

If the database server finds a PSU with space available, it adjusts the MAXSIZE value for the partially full PSU and any subsequent unused PSUs on the full file system to their current sizes (the INITSIZE value for unused PSUs). Then it continues writing the data to the next PSU with available space, issuing a warning message to indicate which file systems ran out of space and which PSU is being used.

However, if no subsequent PSUs have space available in the segment, the operation terminates with an out-of-space error, and no changes are made to the MAXSIZE values.

Whenever a dynamic adjustment of MAXSIZE values occurs, the following conditions apply to the affected segment and PSUs:

- Each new adjusted MAXSIZE value is reflected in the MAXSIZE and USED columns in the **RBW_STORAGE** table and the **TOTALFREE** column in the **RBW_SEGMENTS** table.
- After a PSU is dynamically resized, it cannot be resized again (manually or dynamically) while it is attached to the same table or index in order to take advantage of space that becomes available on the previously full file system. (If space becomes available, the database server can allocate that space to PSUs subsequent to the then-current PSU.)
- If the table that owns a segment is dropped, but its associated segments are retained, the effective size of all PSUs reverts to the MAXSIZE values prior to the dynamic adjustment.

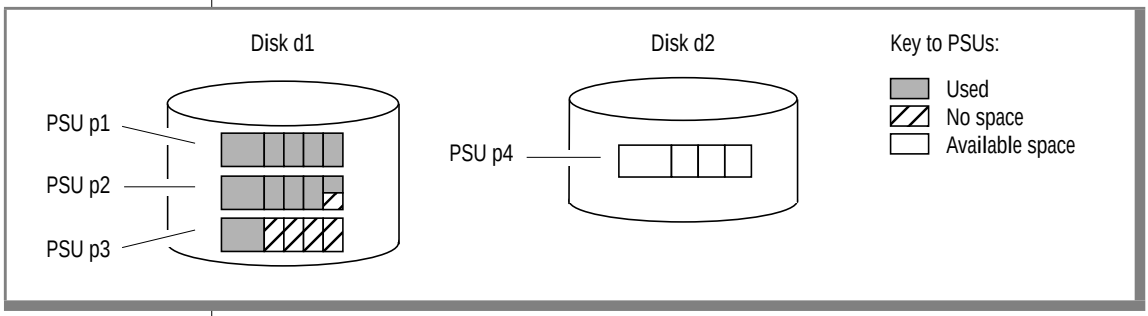
Example

This example illustrates how MAXSIZE is adjusted for the current and subsequent PSUs in a segment when the file system containing the PSUs runs out of space before the PSUs are full.

Assume a table contains a segment that consists of multiple PSUs, **p1**, **p2**, **p3**, and **p4**, across multiple file systems on disks **d1** and **d2**. PSU **p1** is full, and data is being written to PSU **p2**—the current PSU—when the file system on disk **d1** fills. PSU **p2** has used only 620 out of 680 blocks, and PSU **p3** is still empty. The database server automatically adjusts the MAXSIZE value for PSU **p2** to `620` and for PSU **p3** to `120` (the INITSIZE value). It writes data to any preallocated INITSIZE blocks on the full file system, in this case PSU **p3**, and then begins writing to the next PSU on a file system with space available, in this case PSU **p4** on disk **d2**.

The following figure and table illustrate this scenario.

Figure 9-2
PSU Size Adjustments



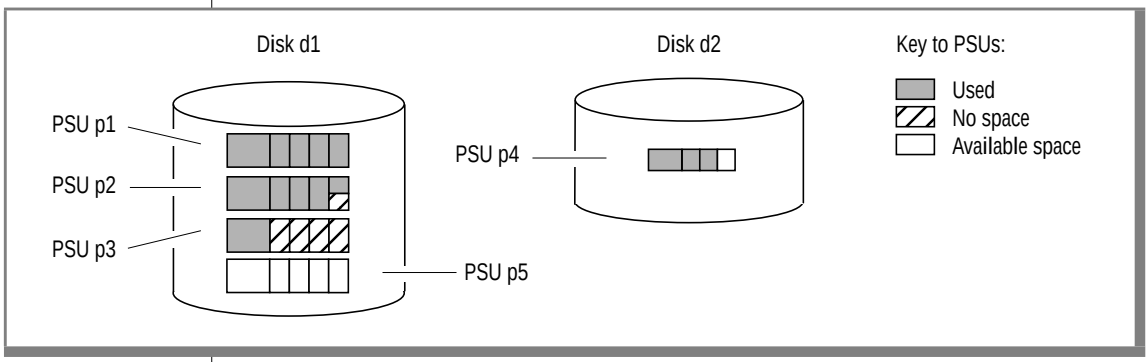
MAXSIZE is dynamically decreased in this example, as the following table illustrates.

PSU	INITSIZE	EXTENDSIZE	Original MAXSIZE	New MAXSIZE
p1	120	100	680	No change
p2	120	100	680	620
p3	120	100	680	120
p4	300	150	750	No change

No new space is ever added to a PSU that precedes the current one. If space later becomes available in the file system on disk **d1**, that space is not used for PSU **p2** or PSU **p3** (while they are attached to the current table or index).

However, if the current PSU is PSU **p4** on disk **d2** and PSU **p5** exists in the file system on disk **d1**, space that becomes available on disk **d2** can be used by PSU **p5** (or any PSU that follows PSU **p4** in the segment storage specification or was added with an ALTER SEGMENT statement after PSU **p4** was defined). The following figure illustrates this scenario.

Figure 9-3
Space Use Between Disks



Altering Segments

You can use segments to distribute data and indexes over multiple drives and to allow continued use of a table when some segments are offline or removed permanently.

Using ALTER SEGMENT

The operations you can perform on segments are as follows:

- Attach or detach a segment from a table or its indexes to add new segments as tables grow and to remove segments when the data in them becomes obsolete (ATTACH, DETACH)
- Take a segment offline to load data into it or to detach it from a table or index, or bring a segment online after it is loaded with data or restored from a backup (OFFLINE, ONLINE)
- Remove all data from a segment, leaving it attached for reuse (CLEAR)
- Change the range specification for segments, to change how data is distributed among segments (RANGE)
- Add PSUs to a segment to accommodate growing tables (ADD STORAGE)

- Change the maximum size, the extend size, or the path of PSUs to control their growth and move them as needed (CHANGE MAXSIZE, CHANGE EXTENDSIZE, CHANGE PATH)
- Rename the segment as needed for easier identification (RENAME)
- Specify a segmenting column to segment a table or index that originally was created in a single default segment and therefore was not defined with a segmenting column (SEGMENT BY)
- Verify that the PSUs that make up the segment are physically intact and determine what is wrong if a segment is damaged (VERIFY)
- Force an undamaged segment that has been marked damaged into an intact state (FORCE INTACT)

You must have DBA authorization or be the creator of a segment to alter it.

Ensuring No Users Are Active

Before using the ALTER SEGMENT statement, ensure that no users are accessing any objects affected by the ALTER SEGMENT operation. Unless you are sure that no users are logged on the database, use either a LOCK DATABASE statement or an ALTER SYSTEM QUIESCE statement. If objects in the database are being accessed, the ALTER SEGMENT operation might fail.

Lock the database with the LOCK DATABASE statement to ensure exclusive access and unlock it with the UNLOCK DATABASE statement when finished, as follows:

```
lock database;
alter segment …;
unlock database;
```

For a slightly less disruptive action than locking the database, suspend all new commands for a database with the ALTER SYSTEM QUIESCE statement and then resume activity on the database, as follows:

```
alter system quiesce;
alter segment …;
alter system resume;
```

The following sections describe the tasks you can perform with the ALTER SEGMENT statement. For syntax descriptions and more detailed information, refer to the *SQL Reference Guide*.

Attaching and Detaching Segments

As tables increase or decrease in size or access patterns change, you can attach or detach segments for the tables or their indexes to meet your needs. You cannot attach segments to tables segmented by hashing.

A segment must exist before it can be attached. Create user-defined segments with a CREATE SEGMENT statement and default segments with a CREATE TABLE or CREATE INDEX statement. Newly attached segments are automatically set to ONLINE mode.

To attach a segment to a table or its B-TREE or TARGET index, supply a valid range specifier based on the data type of the segmenting column. To attach a segment to a STAR index, the range is optional but must be based on the segment name and row numbers (**RBW_ROWNUM** pseudocolumn) of the segmenting column for the referenced (dimension) table. The segmenting column of the referenced table is defined in the CREATE TABLE statement.

Detaching a segment removes the segment from the table or index and removes all the row data or index entries in the segment. User-created segments are not deleted but remain empty and available for reattachment to another object. Default segments are deleted.

To move a segment from one end of the segment range to the other—for example, if you are dropping the oldest data in a time-cyclic database and want to reuse the segment for the new data— detach the segment and attach it with a new range specification.

Moving Entire Segments

You can move an entire segment from one location to another; for example, from one disk to another and between disk and optical storage. Use the ALTER SEGMENT…MIGRATE TO statement. For the syntax of ALTER SEGMENT and for details on the MIGRATE TO clause, refer to the *SQL Reference Guide*.

Specifying a Segmenting Column

If a table or index resides in a single segment and no segmenting column was specified when the table or index was created, you must define a segmenting column before you can attach additional segments.

You cannot specify a segmenting column to change the segmenting column of a table or index or to assign a segmenting column to a table or index that is segmented by hash values.

Specifying a Range

To change the range of data or index entries that reside in a segment, specify a new range of data values or row numbers for the segment. Range modifications cannot leave any gaps or overlaps in the segmentation ranges for the table or index nor can they require the movement of any row data or index entries from one segment to another.

Taking a Segment Offline or Online

Taking a segment offline makes the segment temporarily unavailable to users while allowing partial access to the rest of the table. Offline mode is useful when you need to load additional data or restore a damaged segment but still want to keep the remainder of the table or index available to users. You must take a segment offline before you can detach it.

You can control how the partial unavailability of a table or index affects users with the SET PARTIAL AVAILABILITY options in the **rbw.config** file. For more information about partial availability, refer to "Partial Availability of Tables and Indexes" on page 2-13.

After you have finished modifying an offline segment, you need to synchronize the segment (the data and index structures) with the rest of the table or index. Use a TMU SYNCH command. For more information about the SYNCH command, refer to the *Table Management Utility Reference Guide*.



Tip: *You cannot take all segments of a table or index offline. At least one segment must remain online. This restriction means you can neither take the last remaining online segment of a multisegment table or index offline nor take the only segment of a single-segment table or index offline.*

Clearing a Segment

Clearing a data segment removes all the data from it, as well as the index entries that reference that data. You can clear a segment only if it is one of multiple data segments attached to a table. You cannot clear index segments.

If aggregate maintenance is enabled, then when a segment is cleared, any precomputed views are updated or rebuilt. For information about aggregate maintenance, see the *Informix Vista User's Guide*.

Renaming a Segment

You can rename a user-created or default segment while it is attached to or detached from a table or index. The primary reason to rename a segment is to give it a meaningful, relevant name.

Changing PSU Sizes

You can change the MAXSIZE and EXTENDSIZE values for a PSU to effectively manage disk storage as database tables, indexes, and access patterns change. For example, if a segment runs out of space, you can change the MAXSIZE value of the last PSU in the segment to allow that segment to continue to grow.

The INITSIZE parameter (which you cannot change) determines how much disk space is allocated when the segment is created. The default value for INITSIZE is 16 kilobytes. The database server allocates additional space only as data is stored in the PSU. It is allocated by the amount specified for the EXTENDSIZE value (rounded up to the nearest 8-kilobyte multiple). The default EXTENDSIZE value is 8 kilobytes. A PSU grows to its MAXSIZE value unless the file system becomes full before the MAXSIZE value is reached. In this case, the system dynamically changes the MAXSIZE value, as described on page 9-26.

Changing PSU Location

You can change the location of a PSU to effectively manage disk storage as database tables, indexes, and access patterns change.

To change the location of a PSU, move it from one physical location to another (that is, change its filename) with an operating-system move or copy operation. Then update the **RBW_STORAGE** system table with the CHANGE PATH option to reflect the new location. The order in which you perform these operations does not matter. You can move the file first and then update **RBW_STORAGE** or vice versa, but prevent user access while the operation is in progress by locking the affected table. If you are moving several PSUs, lock the database to ensure that you can complete all the changes to **RBW_STORAGE** before users attempt to access PSUs in transition.

Use caution when moving files, especially files of the same size and with similar names. If you do not correctly correlate the pathname and object, system corruption might result. If you change the path of the wrong object, but realize your mistake before running the TMU or making any modifications (INSERT, UPDATE, or DELETE) to that object, you can reverse the operation by executing this option again with the correct name.

You must also use the CHANGE PATH command if you are moving or copying a table or database containing full pathnames. For example, if you want to move a table in a database, you can copy all the PSUs in the table to the new location and then use this option for each relocated PSU to specify the new pathname for each PSU.

Verifying a Segment

You can determine whether a segment is damaged—that is, PSUs in the segment cannot be opened—and what the damage is with the ALTER SEGMENT…VERIFY option. After you have repaired the damage, use the VERIFY option as part of the recovery process to make sure the damage is fixed and the PSUs can be accessed. The VERIFY option does not actually repair damage, nor does it bring a segment online.

Forcing a Segment into an Intact State

Sometimes a segment is marked damaged after minor or transient access errors have occurred, and the segment cannot be opened even though the PSUs are physically intact. For example, if the file system is not mounted when a query is issued, or if transient NFS errors occur on UNIX, the inaccessible segment is marked as damaged even though there is no actual damage.

If you are certain that the PSUs are physically intact, you can quickly mark the segment intact with the FORCE INTACT option. This option marks the segment intact in the INTACT column of the **RBW_SEGMENTS** table but does not examine each PSU for physical damage. (The VERIFY option examines each PSU for physical damage and takes more time to execute.)



Warning: *Use the FORCE INTACT option only when you know the segment is undamaged. If you are uncertain, use the VERIFY option.*

Recovering a Damaged Segment

Occasionally an operation on a table or index fails with a message that a segment is damaged, or the ALTER SEGMENT...VERIFY statement might indicate that a segment is damaged. The damage is to a PSU within the segment, causing the PSU, the segment, and the table or index to be placed in a damaged state the first time that PSU is accessed.

A PSU is marked damaged (not intact) in the **RBW_STORAGE** system table for any of the following reasons:

- File cannot be found.
- Permissions are inadequate.
- I/O errors or other operating-system errors are encountered trying to open and read a PSU. This type of error might indicate hardware failure that has corrupted the database.

NFS errors of a transient nature are caused by network loading. This type of error generally does not indicate database corruption.

If a segment is damaged, the table or index cannot be accessed while the damaged segment is online. To provide users with partial access to a multisegment table or index with a damaged segment, take the segment offline while you fix the problem.

To recover a damaged segment, you must determine which segment is damaged and what the damage is, repair the damage, and then complete the recovery process.

To recover a damaged segment

1. Determine which segment is damaged.

   You can determine which segment is damaged either from the error message, which is the easier alternative, or from the system tables if the message is not available. **RBW_TABLES**, **RBW_INDEXES**, and **RBW_SEGMENTS** each contain a column named **INTACT**. An **INTACT** value of `N` indicates that the table, index, or segment in question is damaged.

   To find a damaged table or index, enter the appropriate query:

   ```
   select name, intact from rbw_tables where intact = 'N';
   select name, intact from rbw_indexes where intact ='N';
   ```

   To find which segment is damaged, enter a query similar to the following one:

   ```
   select name, tname, iname, intact from rbw_segments
      where intact = 'N';
   ```

2. If you are certain the PSU is not damaged but was marked as such for minor or transient access errors, you can use the ALTER SEGMENT…FORCE INTACT statement (for both online and offline segments) to mark the segment and table or index intact:

   ```
   alter segment seg_name of table table_name force intact;
   alter segment seg_name of index index_name force intact;
   ```

   This statement does not verify the accessibility or integrity of the PSU, but it permits you to avoid a time-consuming verification when you know a PSU has no actual damage.

   If you use the FORCE INTACT option, skip to step 7. Otherwise, continue with step 3.

3. If the damaged segment is part of a multisegment table or index, decide whether to take the segment offline to provide users with partial availability of the table or index.

4. Determine the cause of the damage with the ALTER SEGMENT…VERIFY statement by entering the appropriate query:

```
alter segment seg_name of table table_name verify;
alter segment seg_name of index index_name verify;
```

   The ALTER SEGMENT…ONLINE statement performs the same verification tasks and returns the same information as the VERIFY option. However, this statement cannot be used on segments already online.

5. After you have determined what the damage is, you need to repair it. Problems such as insufficient permission on PSU files or files that are in the wrong directory are easy to remedy. Other problems can be more difficult or impossible to fix, so you might need to restore the segment from a backup or perhaps restore the full database.

6. To confirm that the damage is repaired, use the statement ALTER TABLE…VERIFY.

7. If the segment is offline, set it to ONLINE mode with the statement ALTER SEGMENT…ONLINE.

Managing Optical Storage

Optical storage devices provide direct-access secondary storage that is faster than tape and less expensive than disk. Optical storage offers you additional flexibility in determining how much data to store for how long. The data stored on optical devices is accessed just like data on magnetic disks. Although the access time is longer, the cost is significantly less. Optical devices are a good choice for infrequently accessed data that you neither want to relegate to tape archives (because it is needed occasionally) nor want to store on magnetic disks (because they are more expensive).

Because the access time is longer, you can specify whether queries and certain commands should wait for or skip data and indexes in optical storage. You can also specify whether STAR indexes that reside partly or entirely in optical storage should be considered when an index is chosen, just as you do for offline segments.

This section discusses optical storage support in terms of:

- Assigning optical storage
- Moving entire segments among various types of storage
- Specifying access behavior when row data and indexes reside on optical storage (OPTICAL_AVAILABILITY parameter)
- Specifying whether STAR indexes that reside on optical storage should be considered when an index is selected (IGNORE_OPTICAL_INDEXES parameter)

Assigning Optical Storage

To place an entire segment or specific PSUs on optical storage devices, specify pathnames that point to an optical device in the CREATE SEGMENT or ALTER SEGMENT statements.

To assign the optical property to a segment, include the OPTICAL ON option:

```
alter segment daily_data1 of table sales optical on;
```

If a segment contains any PSUs that reside on an optical storage device, the segment is considered an *optical segment*. You can determine whether a specific segment is an optical segment by using the segment name or segment ID to check its value in the **OPTICAL** column in the **RBW_SEGMENTS** system table:

```
select optical from rbw_segments where name = 'seg_name';
```

You can also search the **RBW_SEGMENTS** system table for optical segments:

```
select name, id from rbw_segments where optical = 'Y';
```

Specifying Access Behavior for Optical Segments

The optical availability option can be set either in the **rbw.config** file or with SET commands. The response to queries or commands that require access to data or indexes in optical segments depends on both the setting for the OPTICAL_AVAILABILITY option and whether the query or command involves read or write operations. Statements affected by this option are:

- Read operations: SELECT and TMU UNLOAD
- Write operations:
  - ALTER TABLE and DROP TABLE
  - CREATE INDEX and ALTER INDEX
  - ALTER SEGMENT
  - INSERT, UPDATE, and DELETE
  - TMU LOAD DATA and REORG

You can determine or verify access behavior for optical segments by checking the OPTICAL_AVAILABILITY entry in the **RBW_OPTIONS** table with a query similar to the following one:

```
select substr(option_name, 1, 30), substr(value, 1, 12)
   from rbw_options
   where option_name like 'OPTICAL%'
      and username = CURRENT_USER;

OPTICAL_AVAILABILITY        WAIT_NONE
```

To specify access behavior with optical segments for all sessions, enter the following line in the **rbw.config** file:

```
OPTION OPTICAL_AVAILABILITY INFO
```

To specify the query behavior for specific sessions, enter a SET command using the following syntax:

```
set optical availability error
```

For a complete list of the optical availability settings, refer to the *SQL Reference Guide*.

Specifying Index Selection with Optical Segments

Whether indexes with PSUs that reside on optical storage are considered when the best index for a query is selected depends on the setting of the IGNORE_OPTICAL_INDEXES option. Suppose a query is processed and an error or warning message indicates that an optical index was accessed, but you know that other fully available but less optimal indexes exist. You can set the IGNORE_OPTICAL_INDEXES option to force the use of an index not residing on optical storage.



Tip: *In most cases, frequently used indexes do not reside in optical segments. Storing an index on slower optical devices defeats the purpose of the index.*

To determine whether indexes with optical segments are considered during index selection, check the IGNORE_OPTICAL_INDEXES entry in the **RBW_OPTIONS** table with a query similar to the following one:

```
select substr(option_name, 1, 30), substr(value, 1, 12)
   from rbw_options
   where option_name like 'IGNORE_OPTICAL%'
      and username = CURRENT_USER;

IGNORE_OPTICAL_INDEXES        OFF
```

To specify the use of indexes in optical segments for all sessions, enter a line in the **rbw.config** file:

```
OPTION IGNORE _OPTICAL_INDEXES OFF
```

To specify the use of indexes that are stored in optical segments for specific sessions, enter a SET command:

```
set ignore optical indexes off;
```

The default setting is ON.

Altering Tables

You can make the following changes to a table with the ALTER TABLE statement:

- Add or drop a column
- Change a column name

- Change a column default value
- Change the specified maximum number of rows (MAXROWS PER SEGMENT and MAXSEGMENTS) in a table
- Change the action taken to maintain referential integrity during delete operations that affect a specified table
- Add or drop a foreign key
- Change the fill factor of a VARCHAR column
- Change the start and step value of a SERIAL column

Informix recommends that you make a backup of a table and its associated indexes before you perform an ALTER TABLE operation on it. Also, whenever you make a change to a table with an ALTER TABLE operation, update CREATE TABLE statements to reflect the changes you made so you can re-create the table from scratch, if necessary.

The following sections describe the tasks you can perform with the ALTER TABLE statement and recovery from an interrupted ALTER TABLE operation. For syntax of this statement, refer to the *SQL Reference Guide*.



Warning: *ALTER TABLE statements are best performed with the IN segment option, which writes the altered table to a new segment. The IN_PLACE option rewrites the table to the same segment. In the event of a failure in the operation, the table is left in disarray. If you want to alter the table in place, as you might for a large table and not enough disk space, make a backup copy first.*

Adding and Dropping Columns

You can add or drop one or more columns with a single ALTER TABLE statement.

When a new column is added with ALTER TABLE...ADD COLUMN, it is added at the end of the table. If multiple columns are added, they are added in the order named. Adding a column does not affect a view because column references are resolved when the view is created.

When a new column is dropped with ALTER TABLE...DROP COLUMN, its data is removed from the table. The operation is not reversible. A column cannot be dropped if it is part of a primary key, part of a foreign key, part of an index key, or referenced by a view.

When you add or drop a column, you can either specify that the changes take place in the existing table location (IN PLACE) or provide another location (one or more segments) in which the altered table is built. If the table is built in another location, the original table is removed when the changes are complete. The advantage to building the table in another location is easier recovery if the operation is interrupted. However, it requires space for two tables. In either case, enough space must be available to hold the modified table.

Changing a Column Name

You can change a column name with an ALTER TABLE…ALTER COLUMN… RENAME statement. The new name must be unique in the table.

Changing the Default Value for a Column

You can change the default value for a column with an ALTER TABLE…ALTER COLUMN… SET DEFAULT statement. The default value is the value that is loaded into a column when the input record was empty or did not contain valid input data. This default value is also used with the TMU Automatic Row Generation option.

For more information about Automatic Row Generation, refer to the *Table Management Utility Reference Guide*.

Changing the MAXSEGMENTS and MAXROWS PER SEGMENTS Values

The MAXSEGMENTS and MAXROWS PER SEGMENT parameters specify the estimated maximum number of rows in the table. These values are used to build a STAR index and to validate segmentation of STAR indexes. To add more rows or segments than these parameters allow, use ALTER TABLE to increase the MAXSEGMENTS and MAXROWS PER SEGMENT values.

If the table is in a default segment, the segment grows as needed to accommodate the extra rows. However, if the table is in a user-defined segment, check the MAXSIZE_ROWS value for that table in the **RBW_TABLES** system table before you change MAXSEGMENTS and MAXROWS PER SEGMENT. If the new value of MAXSEGMENTS multiplied by the new value of MAXROWS PER SEGMENT is greater than MAXSIZE_ROWS, the table outgrows the segment before you reach the new limit. At this point, you must use ALTER SEGMENT to increase the MAXSIZE value of the segment.

For information on setting these parameters and the possible cause of errors involving maximum number of rows per segment, refer to "Setting the MAXSEGMENTS and MAXROWS PER SEGMENT Options" on page 5-14.

Changing the Way Referential Integrity Is Maintained

You can change the way referential integrity is maintained for a given table with an ALTER TABLE…ALTER COLUMN…ON DELETE statement. If a foreign key in a table references another table, the ON DELETE clause lets you specify whether:

- A delete action in the referenced table cascades into the referencing table.
- No delete action is performed on either table.

For more information about referential integrity and cascaded deletes, refer to "Delete Operations and Cascaded Deletes" on page 2-41.

Changing the Data Type for a Column

You might want to change a column data type; for example, to change a TINYINT to a SMALLINT column or to change a numeric external to a date data type.

If a CHAR column has values of widely differing lengths, you can decrease its storage space by changing the data type to VARCHAR. Although a VARCHAR column might use less storage space, an update to a VARCHAR column can sometimes cause the row to be stored less efficiently if the new row is longer than the old row. This situation slows down all subsequent accesses to that row. Informix recommends using the CHAR data type for columns that are frequently updated.



Tip: *A VARCHAR column header is 2 bytes longer than that of a CHAR column. Do not use VARCHAR for short columns (particularly those shorter than 6 bytes) or for columns with data that is all the same length, such as a telephone number field.*

To change the data type of a column

1. Use ALTER TABLE to add a new column of the desired data type. The column must have a unique name not already used in the table.
2. Use an UPDATE statement to copy the data from the old column to the new column.
3. Use ALTER TABLE to drop the old column.
4. Use ALTER TABLE to rename the new column.

Adding and Dropping Foreign Keys

Sometimes a referenced table must be added to or dropped from a schema. There are many reasons why this might occur. For example, a reorganization of the sales force might require the addition of a new district table to a sales database, or a merger might necessitate a new type of product that is different enough to warrant a new table. You might want to associate these new tables with another table by adding a foreign key reference from an existing table. Use an ALTER TABLE...ADD FOREIGN KEY operation on the existing table. With the ALTER TABLE...ADD FOREIGN KEY specification, you can also add a new constraint to the table, as long as the new constraint name is unique.

The requirements for adding a foreign key are as follows:

- The referenced table must exist.
- A primary key index must exist on the referenced table (This index is created automatically when the table is created.)
- The column names named for the foreign key must exist.
- The columns must be declared NOT NULL.
- The data type and length of the referenced columns must exactly match those in the primary key of the referenced table.
- The data must not violate referential integrity.

The data is retrieved to check for referential integrity violations when you add the foreign key, if data exists. If a referential integrity violation occurs, the ALTER TABLE operation terminates with an error, and the table is restored to its original state. For large tables, the referential integrity checking can take some time.

Similarly, suppose a referenced table is no longer needed and you want to drop both the table and the foreign key reference to it. Use an ALTER TABLE... DROP CONSTRAINT operation on the existing table. No STAR index must exist on the constraint. Otherwise, the ALTER TABLE statement fails.

For the complete syntax and usage of the ALTER TABLE statement, refer to the *SQL Reference Guide*.

Changing the Fill Factor for a VARCHAR Column

To adjust the fill factor for a VARCHAR column, use the ALTER TABLE CHANGE FILLFACTOR statement. This statement does not take effect until you execute an ALTER TABLE statement to add or drop a column when the whole table is rewritten using the new fill factor.

For more information, refer to "Setting the VARCHAR Column Fill Factor" on page 10-29.

Changing the Start and Step Values of a SERIAL Column

To change the start or step values of a SERIAL column, use the ALTER TABLE... ALTER COLUMN...SET SERIAL statement. You can specify a new start value (START) or a new step value (STEP).

When you change the start value, the current largest serial value (stored internally by the system) changes to the new start value minus the current step value.



Warning: *If the new start value is less than the current largest serial value in the table, the automatically generated serial value might not be unique.*

When you change the step value, the current largest serial value remains the same.

If you change both the start and step value, change the step value first to avoid the generation of ambiguous serial values.

Recovering from an Interrupted ALTER TABLE Operation

Occasionally an ALTER TABLE operation is interrupted before it completes. Several possible recovery options exist. The option you choose depends both on the cause of the interruption and whether the table was being altered with the IN PLACE option.

Recovering the Table

After dealing with the cause of the interrupt, you have three choices for recovering the table, depending on the state of the table when the interrupt occurred:

- You can resume the alter operation and let it run to completion. This alternative is useful when much of the work has already been done.
- You can reset the table to its original state and reissue the original ALTER TABLE statement. This alternative makes sense when little of the work has been done. However, if you are using the IN PLACE option, you cannot reset the table. You must either resume the operation or restore the table from a backup.
- You can restore the database or just the segments containing the table from a backup. If the backup was current, the table is restored to its state before the ALTER TABLE statement was issued. If the backup is not current, the restored table might not reflect its latest state.



Warning: *If you are altering the table with the IN PLACE option, you should have a current database backup before beginning an ALTER TABLE statement.*

Interruptions: Causes and Prevention

Interruptions can result from:

- Privilege violations

  To prevent interruptions from privilege violations, make sure the user executing the operation has the necessary file system read and write privileges, as well as the required database authority and object privileges.

- Out-of-space errors

  The ALTER TABLE statement calculates how much space is required for the operation. It compares this requirement with the maximum space defined for the segment (the sum of the PSU MAXSIZE values) and does not begin the operation if the requirement exceeds the defined space.

  To prevent interruptions from out-of-space errors, carefully estimate the amount of space required for the altered table and verify that the required space is available before beginning an ALTER TABLE statement.

  Consider the following issues to prevent out-of space errors:

  - The maximum space defined is not necessarily the space allocated. Even though the ALTER TABLE statement calculates that enough space is available, some of that space might not be available. Only the space specified as the INITSIZE value for each PSU is actually allocated when the segment is created.
  - Because the VARCHAR column fill factor is an estimate set by the database administrator, allow extra space for overflows when setting the MAXSIZE value for a segment that contains tables with VARCHAR columns.

- Cancel (CTRL-C) or **kill** commands, system failures, or power failures

  Preventing these types of interruptions is more difficult. Although you cannot anticipate every situation, planning can help you avoid these types of interruptions and deal effectively with them when they happen. Always keep regular backups of your system. Have procedures in place to restore your system so recovery is easy and predictable in the event of a catastrophic failure. Avoid canceling long LOAD operations unless absolutely necessary, especially if you have to force a cancel using an operating-system utility (for example, **kill -9** on UNIX or **pview on** Windows). These operations can leave the data in an inconsistent state.

Copying or Moving a Database

To make a copy of a database for training or testing or to move a database to a new location, you can use combinations of SQL statements and operating-system and TMU commands.

The **rb_cm** utility facilitates the movement of data among databases. For more information on the **rb_cm** utility, refer to the *Table Management Utility Reference Guide*.



Warning: *It is safer to move a database without a version log. If your database contains a version log, drop it before moving the database, and re-create it after the move is complete.*

Full Versus Relative Pathnames

Because either full or relative pathnames can be stored in the system tables, simply copying the files to a new location does not always ensure that the pathnames point to the copy and not the original database. A full pathname begins with a slash (/) on UNIX or a back slash (\) on Windows. A relative pathname is any pathname that does not begin with a slash. The database server constructs relative pathnames relative to the **RB_PATH** environment variable. (If **RB_PATH** is not explicitly defined, it is implicitly defined by the logical database name in the **rbw.config** file.)

The system tables contain only relative pathnames when both of the following conditions are true:

- Each pathname supplied for all PSUs in all CREATE SEGMENT statements present in the database is specified as a relative pathname.
- Each location specified with OPTION DEFAULT_DATA_SEGMENT and OPTION DEFAULT_INDEX_SEGMENT entries in the **rbw.config** file is specified as a relative pathname.

Conversely, the system tables contain some full pathnames when either of the following conditions is true:

- The pathname for any PSU in a CREATE SEGMENT statement is not a relative pathname.
- The pathname for either OPTION DEFAULT_DATA_SEGMENT or OPTION DEFAULT_INDEX_SEGMENT is not a relative pathname.

To determine whether the database uses full pathnames:

UNIX

```
select segname, pseq, location from rbw_storage
   where substr(location, 1, 1) = '/';
```

♦

WIN NT/2000

```
select segname, pseq, location from rbw_storage
   where substr(location, 1, 2) = ':'
      and (substr(location, 1, 3) = '\' ;
```

♦

This statement returns the names of all PSUs that use full pathnames.

After you determine whether the database to be copied or moved contains any full pathnames, use the following procedures to ensure a successful copy or move operation.



Tip: *A database built on a specific platform can be copied only to other locations of the same platform type. For example, a database built on a Solaris platform cannot be copied or moved to an HP 9000 platform or vice versa. To copy or move a database from one platform to another, use the TMU UNLOAD…EXTERNAL operation. Then rebuild the database on the new platform using the data and the CREATE TABLE and LOAD DATA statements that the UNLOAD EXTERNAL operation generates.*

Copy7ing a Database That Contains Only Relative Pathnames

A database that contains only relative pathnames is the easiest kind to copy:

1. Verify that the new location contains enough space to hold the database and that the **redbrick** user can write to the new location.
2. As the **redbrick** user, make a new directory for the database in the new location.
3. Copy the contents of the existing database directory to the new directory.
4. Add a logical name for the copied database to the **rbw.config** file.

Copying a Database That Contains Full Pathnames

If the database contains any full pathnames, use one of the following methods to copy it to a new location.

Method 1

This method is preferred because there is less chance of entering the wrong filenames or pathnames, which might result in unexpected database corruption:

1. Use the TMU UNLOAD...EXTERNAL command to unload the database records to tape or disk. (The TMU also creates the CREATE TABLE, CREATE INDEX, and LOAD DATA statements needed to rebuild the tables and reload the data.)
2. Verify that the new location contains enough space to hold the database and that the **redbrick** user can write to the new location.
3. As the **redbrick** user, make a new directory for the database in the new location.
4. Use the CREATE TABLE and CREATE INDEX statements generated in step 1 to re-create the tables and indexes in the new directory.
5. Use the LOAD DATA statements generated in step 1 to reload the new tables with the data.
6. Add a logical name for the copied database to the **rbw.config** file.



Tip: *You can perform a variation of this procedure using the* ***rb_cm*** *utility. The* ***rb_cm*** *utility pipes UNLOAD output to LOAD input, allowing you to move table data over a network without ever writing to tape or to disk. For more information about the* ***rb_cm*** *utility, refer to the "Table Management Utility Reference Guide."*

Method 2

This method is riskier because there are more opportunities for error in entering pathnames:

1. Verify that the new location contains enough space to hold the database and that the **redbrick** user can write to the new location.
2. As the **redbrick** user, make a new directory for the database in the new location.
3. Copy all files from the existing database directory to the new directory.
4. Copy all files with full pathnames to the new directory.
5. As a database user with DBA authorization or the creator of the segment, use the ALTER SEGMENT…CHANGE PATH statement to change the path for each segment that uses a full pathname to the new pathname (the name of the copy).
6. Add a logical name for the copied database to the **rbw.config** file.

Moving a Database That Contains Only Relative Pathnames

A database that contains only relative pathnames is the easiest kind to move:

1. Verify that the new location contains enough space to hold the database and that the **redbrick** user can write to the new location.
2. As the **redbrick** user, make a new directory for the database in the new location.
3. Copy all files from the existing database directory to the new directory and then delete the original files.
4. Change the logical database name in the **rbw.config** file to the new location.

Moving a Database That Contains Full Pathnames

If the database contains any full pathnames, use the following method to move it to a new location:

1. Verify that the new location contains enough space to hold the database and that the **redbrick** user can write to the new location.
2. As the **redbrick** user, make a new directory for the database in the new location.
3. Use the **cp** command to copy all files from the existing database directory to the new directory and then delete the original files.
4. If you want to leave the files named with full pathnames in their current location, you are finished. Go to step 7.
5. If you want to move any files named with full pathnames to a new location, copy each file from its existing location to its new location and then delete the original file.
6. For each file that you moved in the previous step, as a user with DBA authorization or the creator of the segment containing that file, use the ALTER SEGMENT…CHANGE PATH statement to change the path to the new location.
7. Change the logical database name in the **rbw.config** file to the new location.

Modifying the Configuration File

The configuration file, **rbw.config**, is created when the database server software is installed, using information provided during the installation procedure. The database server and the TMU use this information.

As the **redbrick** user, you can edit the **rbw.config** file (using any standard text editor) as conditions at your site change. Changes you make to this file have various effects on the processes that use it.



Tip: *In the **rbw.config** file, parameters preceded by TUNE affect performance. The parameters preceded by OPTION affect behavior.*

For a description and example of the **rbw.config** file and a table that lists which parameters can also be set with SQL statements or TMU SET commands, refer to Appendix B, "Configuration File."

UNIX

To determine what other actions you must take, refer to the following table.

Change	Action Required
SERVER SHMEM MAPFILE	Stop and restart warehouse daemon. Do not change while warehouse daemon is running.
MAX_SERVERS MAX_ACTIVE_DATABASES PROCESS_CHECKING_INTERVAL	■ Stop warehouse daemon. ■ Use **ipcrm** to remove prior memory segment or reboot. (This step is necessary only after severe errors occur.) ■ Restart warehouse daemon.
CLEANUP_SCRIPT LOGFILE_SIZE QUERYPROCS TOTALQUERYPROCS ADVISOR_LOGGING READER_PRIORITY CHECK_REPORT_FILE_PERMISSIONS GRANT_TEMP_RESOURCE_TO_ALL DEFAULT_SEGMENT_SIZE DEFAULT_PSU_EXTEND_SIZE PRECOMPUTED_VIEW_MAINTENANCE PRECOMPUTED_VIEW_MAINTENANCE_ON_ERROR IDLE_TIMEOUT	Stop and restart warehouse daemon. Do not change while warehouse daemon is running.
SERVER_NAME MESSAGE_DIR LOCALE	Do not change.
INTERVAL	Stop and restart the database server monitoring daemon (**rbw.servermon**).

(1 of 3)

Change	Action Required
FIRST_DAYOFWEEK COUNT_RESULT CHECK_TABLE_INDEX_DIRECTORY UNIFORM_PROBABILITY_FOR_ADVISOR EXPORT_DEFAULT_PATH EXPORT_DELIMITER EXPORT_MAX_FILE_SIZE GROUP QUERY_MEMORY_LIMIT ROWCOUNT INFO_MESSAGE_LIMIT RBW_LOADINFO_LIMIT IGNORE_OPTICAL_INDEXES OPTICAL_AVAILABILITY FILE_GROUP ROWS_PER_SCAN_TASK ROWS_PER_FETCH_TASK ROWS_PER_JOIN_TASK FORCE_SCAN_TASKS FORCE_FETCH_TASKS FORCE_JOIN_TASKS FORCE_HASHJOIN_TASKS FORCE_AGGREGATION_TASKS PARTIONED_PARALLEL_AGGREGATION CROSS_JOIN ARITHABORT ALLOW_POSSIBLE_DEADLOCK DEFAULT_DATA_SEGMENT DEFAULT_INDEX_SEGMENT TEMPORARY_DATA_SEGMENT TEMPORARY_INDEX_SEGMENT ADVISOR_LOGGING PRECOMPUTED_VIEW_QUERY_REWRITE USE_INVALID_PRECOMPUTED_VIEWS SEGMENTS IGNORE_PARTIAL_INDEXES PARTIAL_AVAILABILITY QUERY_TEMPSPACE parameters RESULT_BUFFER RESULT_BUFFER_FULL_ACTION VERSIONING TRANSACTION_ISOLATION_LEVEL	No action needed. Current database server processes do not recognize, but new processes do.

(2 of 3)

Change	Action Required
TMU_ROW_MESSAGES TMU_BUFFERS TMU_OPTIMIZE TMU_CONVERSION_TASKS TMU_INDEX_TASKS TMU_SERIAL_MODE AUTOROWGEN TMU_VERSIONING TMU_COMMIT_RECORD_INTERVAL TMU_COMMIT_TIME_INTERVAL TMU_MAX_TASKS TMU_INPUT_TASKS TMU_MMAP_LIMIT RANDOM_SAMPLE_MARGIN ADVISOR_SAMPLE_SIZE	No action needed. Current TMU processes do not recognize, but new processes do.
FILLFACTOR parameters INDEX_TEMPSPACE parameters PASSWORD parameters	No action needed. Current database server and TMU processes do not recognize, but new processes do.
ADMIN parameters REPORT_INTERVAL	No action needed. Current database server and TMU processes do not recognize, but new processes do.
Other ADMIN parameters	Stop and restart warehouse daemon or use the ALTER SYSTEM statement to restart logging and ADMIN processes.

(3 of 3)

◆

WIN NT/2000

To determine what other actions you must take, refer to the following table.

Change	Action Required
SERVER MAX_SERVERS CLEANUP_SCRIPT LOGFILE_SIZE QUERYPROCS TOTALQUERYPROCS ADVISOR_LOGGING UNIFIED_LOGON MAX_ACTIVE_DATABASES PROCESS_CHECKING_INTERVAL READER_PRIORITY CHECK_REPORT_FILE_PERMISSIONS GRANT_TEMP_RESOURCE_TO_ALL DEFAULT_SEGMENT_SIZE DEFAULT_PSU_EXTEND_SIZE PRECOMPUTED_VIEW_MAINTENANCE PRECOMPUTED_VIEW_MAINTENANCE_ON_ERROR IDLE_TIMEOUT	From the Control Panel, stop the warehouse service and then restart it. Do not change while warehouse service is running.
SERVER_NAME MESSAGE_DIR LOCALE	Set during installation. Do not change.

(1 of 3)

Change	Action Required
FIRST_DAYOFWEEK COUNT_RESULT CHECK_TABLE_INDEX_DIRECTORY UNIFORM_PROBABILITY_FOR_ADVISOR EXPORT_DEFAULT_PATH EXPORT_DELIMITER EXPORT_MAX_FILE_SIZE GROUP QUERY_MEMORY_LIMIT ROWCOUNT INFO_MESSAGE_LIMIT RBW_LOADINFO_LIMIT IGNORE_OPTICAL_INDEXES OPTICAL_AVAILABILITY FILE_GROUP ROWS_PER_SCAN_TASK ROWS_PER_FETCH_TASK ROWS_PER_JOIN_TASK FORCE_SCAN_TASKS FORCE_FETCH_TASKS FORCE_JOIN_TASKS FORCE_HASHJOIN_TASKS FORCE_AGGREGATION_TASKS PARTIONED_PARALLEL_AGGREGATION CROSS_JOIN ARITHABORT ALLOW_POSSIBLE_DEADLOCK DEFAULT_DATA_SEGMENT DEFAULT_INDEX_SEGMENT TEMPORARY_DATA_SEGMENT TEMPORARY_INDEX_SEGMENT OPTION ADVISOR_LOGGING PRECOMPUTED_VIEW_QUERY_REWRITE USE_INVALID_PRECOMPUTED_VIEWS SEGMENTS IGNORE_PARTIAL_INDEXES PARTIAL_AVAILABILITY QUERY_MEMORY_LIMIT QUERY_TEMPSPACE parameters RESULT_BUFFER RESULT_BUFFER_FULL_ACTION VERSIONING TRANSACTION_ISOLATION_LEVEL UNIFORM_PROBABILITY_FOR_ADVISOR	No action needed. Current database server threads do not recognize, but new ones do.

Change	Action Required
TMU_ROW_MESSAGES TMU_BUFFERS TMU_OPTIMIZE TMU_CONVERSION_TASKS TMU_INDEX_TASKS TMU_SERIAL_MODE AUTOROWGEN TMU_VERSIONING TMU_COMMIT_RECORD_INTERVAL TMU_COMMIT_TIME_INTERVAL TMU_MAX_TASKS TMU_INPUT_TASKS TMU_MMAP_LIMIT RANDOM_SAMPLE_MARGIN ADVISOR_SAMPLE_SIZE	No action needed. Current TMU processes do not recognize, but new ones do.
FILLFACTOR parameters INDEX_TEMPSPACE parameters PASSWORD parameters	No action needed. Current database server threads and TMU processes do not recognize, but new ones do.
ADMIN parameters REPORT_INTERVAL Other ADMIN parameters	No action needed. Current threads do not recognize, but new ones do. From the Control Panel, stop the warehouse service and then restart it or use the ALTER SYSTEM statement to restart logging and ADMIN threads.

(3 of 3)

♦

The **RBW_OPTIONS** system table lists current values for all parameters that you can change. It is updated whenever a SET command changes a value during a session. The values displayed apply to the current session. For more information, refer to Appendix C, "System Tables and Dynamic Statistic Tables."

Monitoring and Controlling a Database Server

Red Brick Decision Server includes tools to help monitor and control the warehouse daemons and database server processes. You can also monitor queries through the USAGE ROUTINE event in the log file. The following sections describe these monitoring tools. For a description of other monitoring and control features, see Chapter 8, "Managing Database Activity in an Enterprise."

UNIX

Monitoring and Controlling a Database Server on UNIX

The following sections discuss how to monitor and control database server daemon processes, monitor user sessions, and use log files to monitor database processes.

Daemon Processes

You can use the following scripts to monitor and control the Red Brick Decision Server daemon processes (**rbwapid**, **rbwlogd**, and **rbwadmd**). The **RB_CONFIG** environment variable must be set to point to the directory that contains the **rbw.config** file to run these scripts or any user-defined scripts that call them.

Script	Description
rbw.start	Starts **rbwapid**, **rbwlogd**, and **rbwadmd** daemons as background processes. You must be logged in as the **redbrick** user to execute **rbw.start**: *redbrick_dir*/bin/rbw.start *config_path RB_HOST*
rbw.show	Displays information about the active **rbwapid**, **rbwlogd**, and **rbwadmd** daemons and their associated database server processes: *redbrick_dir*/bin/rbw.show
rbw.stop	Stops **rbwapid**, **rbwlogd**, and **rbwadmd** daemons. You must be logged in as the **redbrick** user to execute **rbw.stop**: *redbrick_dir*/bin/rbw.stop *RB_HOST*

For more information about running these scripts and automatic startup procedures, refer to the *Installation and Configuration Guide*.

If **rbwapid** or **rbwlogd** goes down while the warehouse daemon is still running, you can type the name of the daemon on a command line, and the warehouse daemon restarts it.

If the administration daemon goes down while the warehouse daemon is still running, restart the administration daemon separately by running the **rbwadmd** executable from the **/redbrick/bin** directory.

To stop the administration daemon, use the ALTER SYSTEM TERMINATE ADMIN DAEMON statement.

Findserver Utility

You can use a utility named ***rbw.findserver*** (**Findserver**) to inquire about a specific user session or all active sessions. For each session it finds, the **Findserver** utility lists the following information:

- User name
- Database path
- Database server process ID
- Date and time that a user session started

The **rbw.findserver** program works in conjunction with a database server monitoring daemon named **rbw.servermon**. If the **Findserver** utility is enabled, the **rbw.servermon** daemon is started automatically by the **rbw.start** script and runs whenever the **rbwapid** daemon is running. The monitoring daemon runs in the background and maintains a private record of information about active database server sessions.

Once started, the **rbw.servermon** daemon runs as long as the **rbwapid** daemon is running. When the **rbwapid** daemon stops, the **rbw.servermon** daemon stops at the next scheduled maintenance update of the monitor log file, as determined by the INTERVAL value. You can use the **rbw.stop** utility to stop the **rbwapid** daemon.

Enabling Findserver

To enable database server monitoring and the **Findserver** utility, add the following line to your **rbw.config** file:

```
RBWMON INTERVAL n
```

The RBWMON keyword enables database server monitoring. The INTERVAL parameter specifies the time in seconds (*n*) between maintenance updates of the monitor log file. Informix recommends a value of 120 seconds.

If the RBWMON keyword is not present in the **rbw.config** file, no database server monitoring is performed, and the **rbw.findserver** utility does not function. You must add the RBWMON keyword. The installation script does not add it automatically.

Using Findserver

The **rbw.findserver** program must be run as the **redbrick** user, and the **RB_CONFIG** environment variable must be set correctly. The syntax is as follows:

```
rbw.findserver [db_username]
```

If you specify *db_username*, only sessions for that user are listed. If you omit *db_username*, all active sessions are listed.

Log Files

Several log files available in the **redbrick** directory allow you to monitor the various database processes.

File	Description
install.log	Contains a software identification number and database server installation date.
rbwapid.log	Records configuration information and starting and stopping of the various warehouse daemons. This file is limited in size by the LOGFILE_SIZE parameter in the **rbw.config** file
rbwacct rbwlog	Nontext and binary files located in the *redbrick_dir*/**logs** directory that contain detailed activity and accounting information. Informix Customer Support also uses these files.

For more information on log files, refer to "Event Logging" on page 8-15.

There are log files associated with the Vista Advisor. For more information about Advisor log files, see "Advisor Log Files" on page 8-29.

WIN NT/2000

Monitoring and Controlling a Database Server on Windows

The Red Brick Decision Server service is created automatically by the installation process when the database server is installed. From then on, you can use the Control Panel Services icon to start and stop the database service. To stop the administration thread, use the ALTER SYSTEM TERMINATE ADMIN DAEMON statement.



Warning: *Close the log viewer utility (**logdview**) before restarting the database service. If you do not close the log viewer, the log daemon continues to use the current active log file, and the per-file limit is not in effect.*

As long as the service is running, all of the threads should be running. If a thread does go down, restart the service from the Control Panel. If the Red Brick Decision Server service is deleted for some reason, you can either run the installation procedure to create a new service automatically or use the Registry editor and the **rbwservice** utility to create a new service manually, as the *Installation and Configuration Guide* describes.

You can use the **rbwshow.exe** script to view the host name, number of active connections, and number of active databases. For other monitoring and control features, see Chapter 8, "Managing Database Activity in an Enterprise."

Several log files available in the **redbrick** directory allow you to monitor the Red Brick Decision Server service and its threads.

File	Description
rbwapid.log	Records configuration information and starting and stopping of the service and its threads. This file is limited in size by the LOGFILE_SIZE parameter in the **rbw.config** file.
rbwacct rbwlog	Contain detailed activity and accounting information. These files are located in the *redbrick_dir***logs** directory. Informix Customer Support also uses them. You can read the **rbwlog** files with the **logdview** utility.
rbwexcept.log	Records any software exceptions. This file is used for debugging purposes only.

For more information on log files, refer to "Event Logging" on page 8-15.

There are log files associated with the Vista Advisor. For more information about Advisor log files, see "Advisor Log Files" on page 8-29.

Determining Version Information

If you need to contact the Informix Customer Support, determine the complete version identification information for the Red Brick Decision Server software before you call.

You can obtain database server version information as follows:

- By viewing the **rbwapid.log** file

  The version is at the beginning of the configuration information that follows the startup times.
- From the copyright banners displayed when a RISQL Entry Tool or RISQL Reporter session is started
- By entering the following SQL statement from any tool that allows direct entry of SQL:

  ```
  select rbw_version from rbw_tables ;
  ```

  You can run a similar query on any table. It returns the version number, with one line for each row in the table, so choose a table with a small number of rows.

Deleting Database Objects and Databases

As user requirements evolve, you might need to remove tables from an existing database or delete an entire database from a Red Brick Decision Server installation. This section contains the following information:

- Dropping database objects (tables, indexes, views, synonyms, segments, macros, roles, and hierarchies) from a database with DROP statements
- Deleting a database



Tip: *To remove a user from a database, execute a REVOKE CONNECT statement.*

Dropping Database Objects

To drop objects from a database, use the appropriate SQL DROP statement:

- DROP INDEX
- DROP HIERARCHY
- DROP MACRO
- DROP ROLE (applicable only with role-based security)
- DROP SEGMENT
- DROP SYNONYM
- DROP TABLE
- DROP VIEW

If the object you are dropping is referenced by another object, you must drop the referencing object first. For example, you must drop a view before you can drop the table referenced by the view. Similarly, if a fact table references a dimension table through a foreign key reference, you must drop the fact table before you can drop the dimension table.

If you are going to drop a table, you do not need to drop its columns or indexes first. To drop columns from a table, use the ALTER TABLE statement.

For complete syntax descriptions, refer to the *SQL Reference Guide*.

Indexes

To drop an index from the database, use the DROP INDEX statement.

You can specify whether to keep or drop any user-defined segments associated with the index. If you keep the segments, they are detached from the table associated with the index and are available for reuse. If you drop the segments, all PSUs in the segments are also removed. The default behavior is to keep user-defined segments. To override the default behavior globally, set the SEGMENTS parameter in the **rbw.config** file or locally for a given index in the DROP INDEX statement.

Default segments are always dropped when the index is dropped.

Hierarchies

To drop a hierarchy, use the DROP HIERARCHY statement. You must drop the hierarchies defined on a table before you can drop the table itself. For information about hierarchies, refer to the *Informix Vista User's Guide*.

Macros

To drop a macro, use the DROP MACRO statement. A temporary macro is also dropped when the database connection or database server session is terminated. After a macro is dropped, references to it are no longer expanded.

Use the PUBLIC and TEMPORARY keywords in a DROP MACRO statement in the same way as in the CREATE MACRO statement.

Roles

You cannot drop the predefined system roles, DBA, RESOURCE, and CONNECT. You can only revoke the role from specific users.

You can define custom, or user-defined, roles in addition to the predefined system roles. To drop a user-defined role, use the DROP ROLE statement. Dropping a role removes the role name and effectively revokes all task authorizations, object privileges, database users, and roles that have been granted to the role. Members of the role no longer have the tasks or privileges of the role, but they might have the same tasks or privileges directly or through a different role.

Segments

A segment is dropped from the database with the DROP SEGMENT statement. A segment must be taken offline (with the ALTER SEGMENT statement) and detached before you can drop it with a DROP SEGMENT statement.

Segments can be dropped automatically when the owning table or index is dropped in either of the following ways:

- By setting the SEGMENTS parameter to DROP, either as an option in the **rbw.config** file or with a SET command from the command line

  For more information about setting the SEGMENTS parameter, refer to "Setting Segment and Partial Availability Behavior" on page 10-22
- For a specific table or index, by including the DROPPING SEGMENTS clause in a DROP INDEX or DROP TABLE statement

When a segment is dropped, all PSUs in the segments are also removed.

Synonyms

To drop a synonym, use the DROP SYNONYM statement. This statement has no effect on the base table.

You can drop a synonym when it is no longer needed. Before dropping a synonym, you must drop any tables or views that reference the synonym or alter those tables or views so that they do not reference the synonym.

Tables

To drop a table from the database, use the DROP TABLE statement. This statement also drops any indexes on the table and removes any privileges or synonyms that reference the table. This statement can be used only for base tables, not for views or synonyms.

You can specify whether to keep or drop any user-defined segments associated with the table. If you keep the segments, they are detached from the table and are available for reuse. If you drop the segments, all PSUs in the segments are also removed. The default behavior is to keep user-defined segments. To override the default behavior globally, set the SEGMENTS parameter in the **rbw.config** file. To override the default behavior locally for a given table, set the DROP TABLE statement.

Unlike user-defined segments, default segments are always dropped when the table is dropped.



Tip: *If the table to be dropped contains one or more damaged segments (as indicated in the* ***RBW_SEGMENTS*** *system table), a DROP TABLE…KEEPING SEGMENTS statement fails. Before you can drop the table, you must detach and drop the damaged segments. However, a DROP TABLE…DROPPING SEGMENTS statement succeeds even when the table contains damaged segments.*

Before dropping a table, you must drop any other tables or views that reference the table or any synonyms defined on that table.

Views

To drop a regular view or a precomputed view from the database, use the DROP VIEW statement. Dropping a view does not affect underlying tables. Before you drop a view, you must drop any views that reference it. For more information on precomputed views, see the *Informix Vista User's Guide*.

Deleting a Database

To delete a database, use the utility **rb_deleter** on UNIX or **dbcreate** on Windows. This utility removes any default files created by a database server in the database directory. This utility is typically executed only by the database administrator and can be executed only as the **redbrick** user.

Any files in the database directory that do not have a default name (that is, any user-specified files) and any directories or segment files that are not in the main database directory are not removed automatically by the utility. Instead, you must remove them manually with operating-system commands.

You do not need to drop tables or other objects within the database before deleting the database. However, if the database contains segments that reside outside the database directory, you can use the DROP TABLE statement with the DROPPING SEGMENTS option to automatically remove any associated PSUs, thereby eliminating the need to remove them manually with the **rm** command on UNIX or **del** command on Windows.

UNIX

To delete a database on UNIX

1. Access the database with DBA authority.
2. Perform one or both of the following steps:
   - Use DROP TABLE…DROPPING SEGMENTS for tables with PSUs outside the database directory.
   - Check the **RBW_STORAGE** system table for any PSUs not located in the database directory or the DEFAULT_DATA_SEGMENT and DEFAULT_INDEX_SEGMENT directories. Record their names for use in step 4.
3. Exit from the database.
4. Log in as the **redbrick** user.
5. Invoke **rb_deleter**, which is located in the **$RB_CONFIG/bin** directory. Enter the following command at the system prompt:

   ```
   $ rb_deleter pathname
   ```

   In the example, *pathname* is a full directory path specification to the database directory, such as **/disk1/db_sales_92**.
6. Use operating-system commands (**rm**, **rmdir**) to remove:
   - Any files or directories remaining in the database directory and then remove the directory.
   - Any segment directories and PSUs that were not in the main database directory. (See step 1.) Check for directories referenced by CREATE SEGMENT statements and for directories named by DEFAULT_DATA_SEGMENT and DEFAULT_INDEX_SEGMENT in the **rbw.config** file.
   - Any spill files that were not cleaned up by the processes that created them. These files might exist in directories defined by the QUERY or INDEX TEMPSPACE DIRECTORY settings in the **rbw.config** file or specified interactively from the command line.
7. Remove obsolete logical database name definitions from the **rbw.config** file.
8. Log out as the **redbrick** user. ♦

WIN NT/2000

To delete a database on Windows

1. Access the database with DBA authority.
2. Perform one or both of the following steps:
   - Use DROP TABLE...DROPPING SEGMENTS for tables with PSUs outside the database directory.
   - Check the **RBW_STORAGE** system table for any PSUs not located in the database directory or the directories **DEFAULT_DATA_SEGMENT** and **DEFAULT_INDEX_SEGMENT**. Record their names for use in step 4.
3. Exit from the database.
4. To delete the database, use the **dbcreate** utility and the following syntax:

```
dbcreate -delete { [-d dirname] | [ -l logical_dbname]}
   [-q]
```

-d *dirname*	Full pathname of the directory containing the database to be deleted
-l *logical_dbname*	Logical name of the database to be deleted, as defined in the **rbw.config** file
-q	Quiet mode, which does not ask you to confirm the deletion of each file

For example, from the system prompt:

```
c:\redbrick_dir\util\service: dbcreate -delete -l aroma
```

5. Use operating-system commands (**del**, **erase**, **rmdir**) to remove:
   - Any files or directories remaining in the database directory. Then remove the directory.
   - Any segment directories and PSUs that were not in the main database directory. (See step 1.) Check for directories referenced by CREATE SEGMENT statements and for directories named by **DEFAULT_DATA_SEGMENT** and **DEFAULT_INDEX_SEGMENT** in the **rbw.config** file.
   - Any spill files that were not cleaned up by the processes that created them. These files might exist in directories defined by the QUERY or INDEX_TEMPSPACE_DIRECTORY settings in the **rbw.config** file or specified interactively from the command line.
6. Remove obsolete logical database name definitions from the **rbw.config** file. ♦

Chapter

10

Tuning a Warehouse for Performance

Specifying Parameters with rbw.config File Entries or SET Commands . 10-6

Setting Temporary Space Parameters. 10-7
Temporary Space Parameters 10-8
How Temporary Space Is Allocated 10-9
Random Directory Sequence 10-9
File Creation and Use 10-9
Full and Out-of-Space Conditions 10-11
TEMPSPACE 10-12
Determining Current Values 10-16
Removing Temporary Files. 10-17
Setting QUERY_MEMORY_LIMIT 10-17

Setting the Result Buffer for Long-Running Queries 10-19
RESULT BUFFER Parameter 10-20
RESULT BUFFER FULL ACTION Parameter 10-21

Setting Segment and Partial Availability Behavior 10-22
Location of Default Segments 10-22
Segment Drop Behavior 10-24
Query Behavior on Partially Available Tables 10-25
Use of Partially Available Indexes 10-28

Setting the VARCHAR Column Fill Factor 10-29
How the Database Server Uses the VARCHAR Fill Factor 10-29
Effect of Fill Factor on Performance 10-30
Monitoring Accuracy of the VARCHAR Fill Factor 10-35
Using CHECK TABLE with the VERBOSE Option 10-35
Obtaining the Current Fill Factor Value 10-37
Modifying the VARCHAR Fill Factor 10-37

Setting the Index Fill Factor 10-38
Finding the Fill Factor Used for a Specific Index 10-41
Deciding Whether to Change Default Fill Factors 10-41
Changing an Index Fill Factor 10-42

Creating Additional Indexes. 10-43

Understanding Query Processing 10-44
Join Algorithms. 10-44
Operator Model. 10-47
Advisor 10-48
B-TREE 1-1 Match 10-48
B-TREE Scan 10-48
Bit Vector Sort 10-49
Check 10-49
Choose Plan. 10-50
Delete 10-50
Delete Cascade. 10-51
Delete Refcheck 10-51
Exchange. 10-52
Execute 10-52
Functional Join. 10-52
General Purpose 10-53
Hash 1-1 Match 10-53
Hash AVL Aggregate 10-53
Insert 10-54
Merge Sort 10-54
Naive 1-1 Match 10-54
RISQL Calculate 10-54
Sample Scan 10-54
Simple Merge 10-55
Sort 1-1 Match 10-55
STARjoin. 10-55
Subquery. 10-56
Table Scan 10-56
TARGETjoin 10-56
TARGET Scan 10-57
Update 10-57
Virtual Table Scan. 10-57
EXPLAIN Statement 10-57

TARGETjoin Query Processing 10-61
How to Use TARGETjoin Processing 10-61
Create TARGET or B-TREE Indexes on Foreign Keys of Fact Table 10-61
Rules for TARGETjoin Query Processing 10-62
Turning Off TARGETjoin Query Processing 10-63
When to Use TARGETjoin Processing. 10-64
Evaluate Query Performance 10-64
Schema Types 10-65
Many STAR Indexes Versus TARGETjoin Processing 10-66
Examples 10-66
Query That Chooses TARGETjoin. 10-67
Reading EXPLAIN Output for a TARGETjoin Query 10-69
STAR and TARGET Plan 10-70
TARGET Only Plan. 10-71
Summary and Recommendations 10-74
Indexes to Create 10-76
Large Dimension Table 10-76
Experiment 10-77

Estimates of Fact Table Selectivity 10-77
Example: Selectivity Estimates for Preloaded Dimensions. . . . 10-79
Scenario 10-79
Selectivity Estimate. 10-80

Using Synonyms to Control Fact-to-Fact Joins 10-81

Making SQL-Based Improvements 10-84
UNION Versus Interdimensional ORs 10-84
Subquery in the FROM Clause Versus Correlated Subquery 10-85

In This Chapter

This chapter describes ways in which you can customize Red Brick Decision Server to improve performance at your site. Because customizing is site specific, only general guidelines are provided in most cases. Informix suggests that you use Red Brick Decision Server initially in its default configuration. As you gain experience with the database server software, the system environment, and the workload at your site, you might find areas in which you want to improve performance.

This chapter includes the following sections:

- Specifying Parameters with rbw.config File Entries or SET Commands
- Setting Temporary Space Parameters
- Setting the Result Buffer for Long-Running Queries
- Setting Segment and Partial Availability Behavior
- Setting the VARCHAR Column Fill Factor
- Setting the Index Fill Factor
- Creating Additional Indexes
- Understanding Query Processing
- TARGETjoin Query Processing
- Estimates of Fact Table Selectivity
- Using Synonyms to Control Fact-to-Fact Joins
- Making SQL-Based Improvements

To turn on statistics reporting, which provides some information useful in analyzing query performance, use the SET STATS command.

For information about improving query performance through parallel processing, refer to Chapter 11, "Tuning a Warehouse for Parallel Query Processing." For information about monitoring, controlling, and tracking resource use of the database server, refer to Chapter 8, "Managing Database Activity in an Enterprise." To customize the database server environment for your site and user community, refer to your operating-system documentation.

Specifying Parameters with rbw.config File Entries or SET Commands

You can set many parameters either in the **rbw.config** file or with SET commands. Entries in the **rbw.config** file affect all sessions for that database server. SET commands affect only the session during which they are executed. You can enter most SET commands interactively anywhere you can enter SQL statements, or you can enter them in the warehouse, database, or user **.rbwrc** files. Enter SET commands for the TMU in the TMU control file for the target activity.

The values in the **rbw.config** file are processed before any **.rbwrc** files are read or before the TMU runs, so the **rbw.config** file settings can be overridden by **.rbwrc** file settings or by TMU control files.

The **RBW_OPTIONS** system table lists all tunable parameters and their current values. The table is updated whenever a SET command changes a value during a user session.



Tip: *The syntax diagrams for SET commands in this chapter include the terminating semicolon (;) required by the TMU, the RISQL Entry Tool, and the RISQL Reporter. Not all SQL entry tools require this terminator.*

Setting Temporary Space Parameters

Index-building operations and complex queries can require large amounts of temporary space to store intermediate results. With the temporary space parameters, you can define the directories to be used when temporary space is needed, the threshold at which intermediate results spill from memory to disk, and the maximum amount of disk space to be used for temporary space. Separate parameters control index-building operations and query operations.

This section covers the following topics:

- Parameters used to control temporary space
- Procedure by which temporary space is allocated
- Syntax for temporary space parameters
- Procedure to determine the current values of these parameters
- Removal of temporary files

For information about how to estimate temporary space requirements, refer to "Estimating Temporary Space Requirements" on page 4-35.

Temporary Space Parameters

The following parameters and corresponding SET commands control the location and use of temporary space.

Figure 10-1
For Index Building Temporary Space

TUNE Parameter and SET Command	Function
TUNE INDEX_TEMPSPACE_DIRECTORY, SET INDEX TEMPSPACE DIRECTORIES*	Specifies temporary space directory or directories to be used by index-building operations. Multiple **rbw.config** file entries can be made, one directory per entry. *
TUNE INDEX_TEMPSPACE_THRESHOLD, SET INDEX TEMPSPACE THRESHOLD	Specifies size at which index-building operations spill from memory to disk.
TUNE INDEX_TEMPSPACE_MAXSPILLSIZE, SET INDEX TEMPSPACE MAXSPILLSIZE	Specifies maximum amount of temporary space that can be allocated to a spill during an index-building operation.
TUNE QUERY_TEMPSPACE_DIRECTORY, SET QUERY TEMPSPACE DIRECTORIES	Specifies temporary space directory or directories to be used by query processing. Multiple **rbw.config** file entries can be made, one directory per entry. *
TUNE QUERY_MEMORY_LIMIT, SET QUERY MEMORY LIMIT	Specifies the maximum memory size for queries, after which they spill from memory to disk.
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE, SET QUERY TEMPSPACE MAXSPILLSIZE	Specifies maximum amount of temporary space that can be allocated to service a spill during a query operation.

* Red Brick Decision Server and the TMU automatically spread temporary space usage evenly across the designated directories.

The QUERY_TEMPSPACE parameters are used whenever queries are processed. They are not used for TMU operations. The INDEX_TEMPSPACE parameters are used whenever indexes are built or modified by CREATE INDEX statements or TMU LOAD DATA or REORG operations.

You can specify values for these parameters in the **rbw.config** file or with SET commands, which you can enter in **.rbwrc** files or anywhere you can enter SQL statements. You can also specify INDEX_TEMPSPACE parameters for TMU operations with SET commands in a TMU control file.

How Temporary Space Is Allocated

The set of directories that make up each temporary space are like a circular buffer, and the set of files created in each directory for a particular query or index-building process are like a slice of the buffer, a slice that can grow to but not exceed MAXSPILLSIZE.

Random Directory Sequence

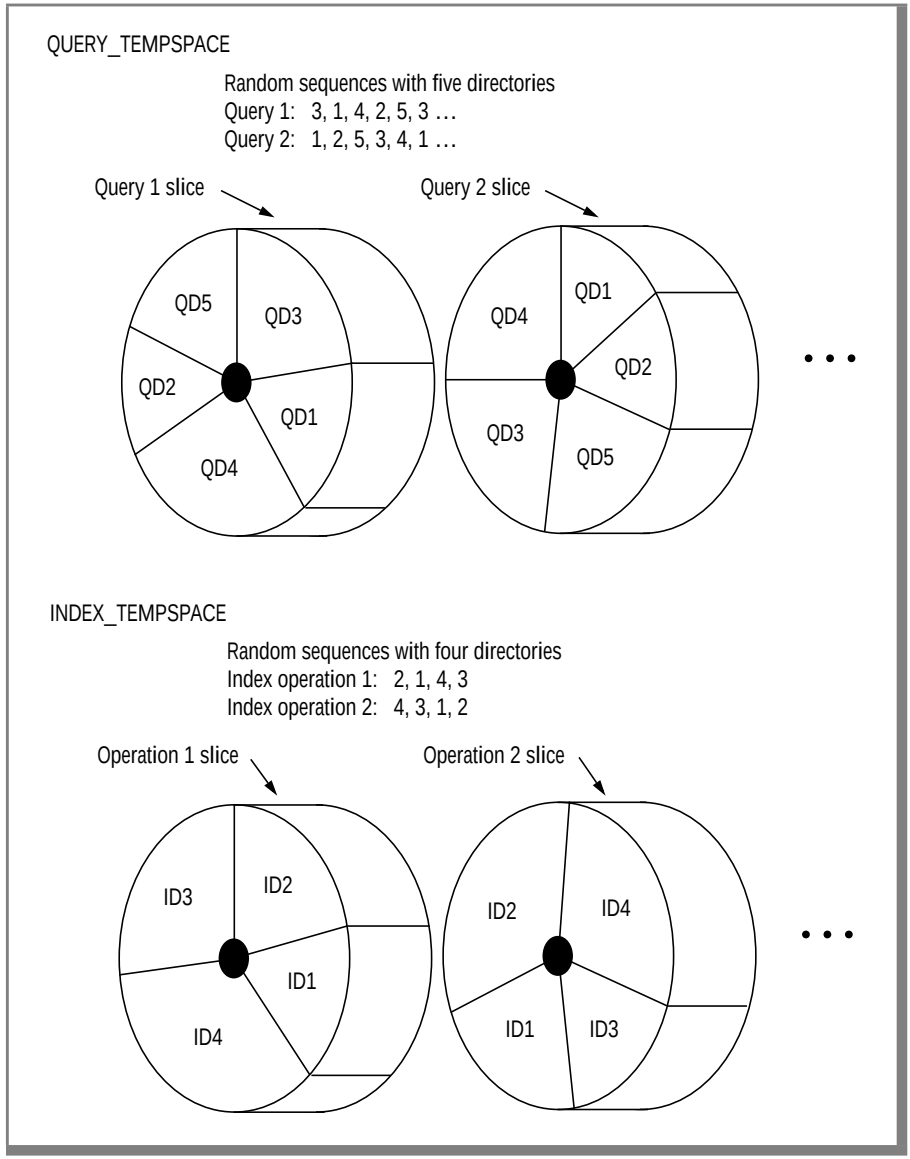
When a process exceeds the threshold size and spills to disk, a random sequence of these temporary space directories is determined for that process. This sequence determines the order in which the directories are used by that process. For example, if five temporary space directories—**d1**, **d2**, **d3**, **d4**, **d5**—are defined, a sequence for one spill operation might be **d3**, **d1**, **d4**, **d2**, **d5**, and the sequence for another concurrent spill operation might be **d1**, **d2**, **d5**, **d3**, **d4**.

The following figure illustrates this concept of the temporary space directories that make up a circular buffer, one for queries and one for index-building operations. Two slices based on random directory sequences are shown for each buffer.

File Creation and Use

For each temporary space slice, one or more files are created in each directory. The first file in the first directory is initialized to 16 kilobytes. If a MAXSPILLSIZE value of less than 8 kilobytes is specified, the first file is initialized to only 8 kilobytes. The remaining files in the slice are initialized with a size of 0.

Figure 10-2
Random Directory Sequence



For both query and index-building spills, the directories are used in the random sequence determined for each spill. However, in directories containing multiple files, the files are used in the following sequences:

- For query temporary directories

  The files are used one per directory in the random sequence, thus distributing the load among the directories, repeating the sequence as needed.

- For index-building directories

  All files in a directory are used, before proceeding to the next directory in the random sequence.

Full and Out-of-Space Conditions

A slice of query or index temporary space is *full* when either of the following conditions is met:

- The sum of the current sizes of all files that make up that slice equals the MAXSPILLSIZE value for that space.
- The slice cannot be further extended (by extending a file) without exceeding the MAXSPILLSIZE value for that space.

A slice of query or index temporary space is *out of space* when the slice is not yet full, but all files that make up that slice are in use, and no more disk space is available to extend the last file in the sequence of files.

Full and out-of-space conditions affect the various operations as follows:

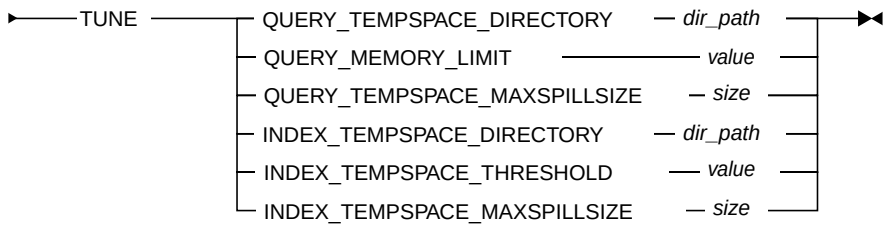
- Queries

  If a slice of query temporary space becomes full or runs out of space, the query is aborted.

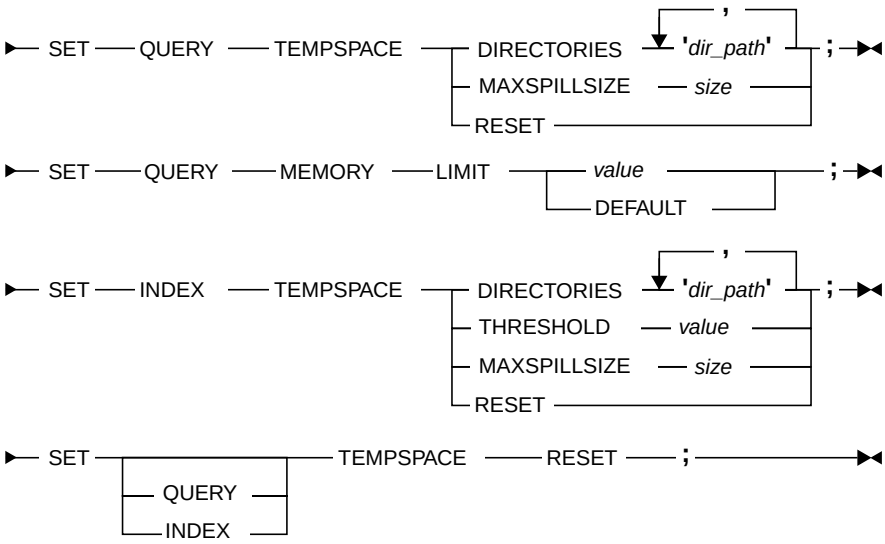
- Indexes:
  - If a slice of index temporary space—online or offline—runs out of space, the operation is aborted.
  - If a slice of *online* index temporary space becomes full, the contents are merged into the index, the temporary space is emptied, and the index build continues, reusing the space.
  - If a slice of *offline* index temporary space becomes full, the operation is aborted.

TEMPSPACE

To specify parameters that apply to all sessions, enter a line in the **rbw.config** file, using the following syntax.



To specify the parameters for specific sessions, enter a SET command using the following syntax.



DIRECTORY *'dir_path'*, DIRECTORIES *'dir_path'*, …	Specifies a directory or a set of directories to be used for temporary files. The *dir_path* variable must be a full pathname. To define a set of directories using entries in the **rbw.config** file, enter multiple lines. The order in which the directories are specified has no effect because they are used in random order (determined internally), and no user control is possible. On UNIX, if no temporary space directories are defined, the default directory is **/tmp**. On Windows, if no temporary-space directories are defined, the default directory is **%TEMP%** or, if not set, **c:\tmp**.
THRESHOLD *value*	Specifies the amount of memory used before writing the intermediate results from an index-building operation to disk. For index-building operations involving multiple indexes, this threshold value is allocated equally among the indexes being built. The size must be specified as either kilobytes (K) or megabytes (M) by appending K or M to the number. No space is allowed between the number and the unit identifier (K, M). For example: 1024K, 500M. The threshold value must be specified before the corresponding MAXSPILLSIZE value is specified. It must precede the MAXSPILLSIZE entry in the **rbw.config** file. A value of `0` causes files to be written to disk after the first 200 rows or index entries. You must specify the units. For example: `0K`, `0M`. For INDEX_TEMPSPACE_THRESHOLD, the default value is 10 megabytes (`10M`).

MAXSPILLSIZE *size*	Specifies the total maximum amount of temporary space per operation. For an index-building operation involving multiple indexes, this space is allocated equally among the indexes being built. For query operations, however, the entire value is allocated to each query and to each of its subqueries, if any. The size must be specified as kilobytes (K), megabytes (M), or gigabytes (G) by appending K, M, or G to the number. No space is allowed between the number and the unit identifier (K, M, G). For example: `1024K`, `500M`, `8G`. The default MAXSPILLSIZE value is 1 gigabyte. The maximum MAXSPILLSIZE value is 2047 gigabytes.
RESET	Resets the query or index TEMPSPACE parameters to the values specified in the **rbw.config** file. If neither QUERY nor INDEX is specified, all TEMPSPACE parameters are reset.
QUERY MEMORY LIMIT *value*	The QUERY MEMORY LIMIT value must be specified as kilobytes (K), megabytes (M), or gigabytes (G) by appending K, M, or G to the number. No space is allowed between the number and the unit identifier (K, M, or G). For example: `2048K`, `500M`, `3G`. The default value of QUERY MEMORY LIMIT is 50 megabytes (`50M`). The range is from 2 megabytes (2M) to 4 gigabytes (4G).

Usage

In addition, use the following guidelines when setting temporary space parameters:

- Always set the QUERY_MEMORY_LIMIT value before setting the QUERY_TEMPSPACE_MAXSPILLSIZE value.
- Never set the QUERY_MEMORY_LIMIT to a value larger than the maximum data segment size allocated to a process by the operating system or larger than the QUERY_TEMPSPACE_MAXSPILLSIZE value.
- In general, smaller QUERY_MEMORY_LIMIT values are better in multiuser environments. A value that is too large can cause excessive paging or higher physical memory usage.
- Use tools or commands to monitor memory usage; for example, **vmstat** and **sar** on UNIX and Performance Monitor (**PerfMon**) or **pstat** and **pview** on Windows.

For additional information about how queries use temporary space, refer to "Setting QUERY_MEMORY_LIMIT" on page 10-17.

For information about temporary space requirements for aggregate maintenance, see the *Informix Vista User's Guide*.

Examples

UNIX

The following example illustrates entries in the **rbw.config** file that apply to all sessions:

```
TUNE QUERY_TEMPSPACE_DIRECTORY /disk1/qtemp
TUNE QUERY_TEMPSPACE_DIRECTORY /disk2/qtemp
TUNE QUERY_TEMPSPACE_DIRECTORY /disk3/qtemp
TUNE QUERY_MEMORY_LIMIT 2M
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE 8G
```

The following example illustrates SET commands that can be used to change parameters for a specific session:

```
SET INDEX TEMPSPACE DIRECTORIES '/disk1/itemp',
   '/disk2/itemp', '/disk3/itemp'
SET INDEX TEMPSPACE THRESHOLD 2M
SET INDEX TEMPSPACE MAXSPILLSIZE 3G
```

♦

WIN NT/2000

The following example illustrates entries in the **rbw.config** file that apply to all sessions:

```
TUNE QUERY_TEMPSPACE_DIRECTORY d:\qtemp
TUNE QUERY_TEMPSPACE_DIRECTORY e:\qtemp
TUNE QUERY_TEMPSPACE_DIRECTORY f:\qtemp
TUNE QUERY_TMEMORY_LIMIT 2M
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE 8G
```

The following example illustrates SET commands that can be used to change parameters for a specific session:

```
SET INDEX TEMPSPACE DIRECTORIES 'd:\itemp',
   'e:\itemp', 'f:\itemp'
SET INDEX TEMPSPACE THRESHOLD 2M
SET INDEX TEMSPACE MAXSPILLSIZE 3G
```

♦

The following example illustrates how to reset the INDEX_TEMPSPACE parameters to the values specified in the **rbw.config** file, leaving the QUERY_TEMPSPACE parameters set to their current values:

```
SET INDEX TEMPSPACE RESET
```

The following example illustrates how to reset all TEMPSPACE parameters to the values specified in the **rbw.config** file:

```
SET TEMPSPACE RESET
```

Determining Current Values

To determine the current values for the QUERY_TEMPSPACE and INDEX_TEMPSPACE parameters, query the **RBW_OPTIONS** system table.

To determine the QUERY_TEMPSPACE parameters in effect for the current session, enter a query similar to the following one:

```
select substr(option_name, 1, 30), substr(value, 1, 40)
   from rbw_options
   where username = CURRENT_USER
      and option_name like 'QUERY?_%' escape '?';
```

Removing Temporary Files

Spill files are usually removed by the database server (**rbwsvr**) or the TMU as soon as possible. However, if the database server or TMU terminates abnormally, it might not be able to remove all spill files before it terminates. To remove any of these old spill files, upon initialization the warehouse daemon (**rbwapid**) on UNIX or the Red Brick Decision Server service on Windows executes a cleanup script specified by the following entry in the **rbw.config** file:

```
RBWAPI CLEANUP_SCRIPT script_name
```

script_name Name of the cleanup script for your system

The default cleanup script, which is *redbrick_dir/bin/rb_sample.cleanup* on UNIX and *redbrick_dir\bin\rbclean.bat* on Windows, removes all files from the QUERY_TEMPSPACE and INDEX_TEMPSPACE directories specified in the **rbw.config** file. If no directories are specified in the file, the script looks for spill files in the **tmp** directory and removes them. The cleanup script does not find and remove spill files from locations specified with a SET command during a database server or TMU session. These files must be removed manually.

Setting QUERY_MEMORY_LIMIT

When setting the QUERY_MEMORY_LIMIT parameter, consider the following:

- Be aware of the amount of physical memory available on your system.
- Do not consistently overcommit memory.
- Know the number of users on the system and the number of users who will be issuing queries during the same time period.
- Keep the paging rate down.

Red Brick Decision Server allocates memory to each query. The size of the memory allocation is allowed to grow from 1 megabyte, which is the default size to which the buffer cache is initialized, to the value specified by the QUERY_MEMORY_LIMIT parameter, after which it spills to disk. The value of the QUERY_MEMORY_LIMIT parameter applies to all users on the system. Therefore, if QUERY_MEMORY_LIMIT is set to 10 megabytes and you have 10 users on your system, the minimum amount of memory used for query processing is 10 times 1 megabyte, or 10 megabyte, and the potential amount of memory that can be used for query processing is 10 times 10 megabytes, or 100 megabytes. Each user's memory consumption grows above 1 megabyte only when a query is issued that requires more than 1 megabyte of memory.

If Red Brick Decision Server runs out of query-processing memory before each user reaches the value specified in the QUERY_MEMORY_LIMIT parameter, the operating system starts swapping memory to disk. Avoid this condition because it slows performance. When you set the parameter QUERY_MEMORY_LIMIT, choose a value high enough that most queries can run in memory but not so high that the operating system has to swap to disk.

Consider trade-offs, however. For example, setting the parameter QUERY_MEMORY_LIMIT to a higher value might avoid spilling to disk in 90 percent of your queries, thus making those queries run many times faster. But the remaining 10 percent might be large queries run simultaneously by several users, potentially overcommitting memory resources, forcing the operating system to swap to disk, and ultimately slowing everyone down. Consider the trade-offs and set the QUERY_MEMORY_LIMIT value accordingly.

The amount of memory used also depends on the number of concurrent users executing queries. Consider this additional factor when you set the value for QUERY_MEMORY_LIMIT. The types of queries that are being executed is also a factor. Queries that browse a small dimension table tend to use small amounts of memory, and queries that join two fact tables tend to use large amounts of memory. It is sometimes difficult to predict what the users will do, but you can find trends by monitoring the database activity.

Setting the Result Buffer for Long-Running Queries

The *result buffer* is an area of temporary space that is used to hold query results that have completed processing on the database server but are waiting for the client to request them. This buffer is necessary because some client tools require user input to retrieve more than a certain amount of data. While a query is processing, a read lock is placed on each of the tables involved in the query. If you have long-running queries, the read locks prevent other users from performing INSERT, UPDATE, DELETE, or LOAD operations on these tables for the duration of the query.

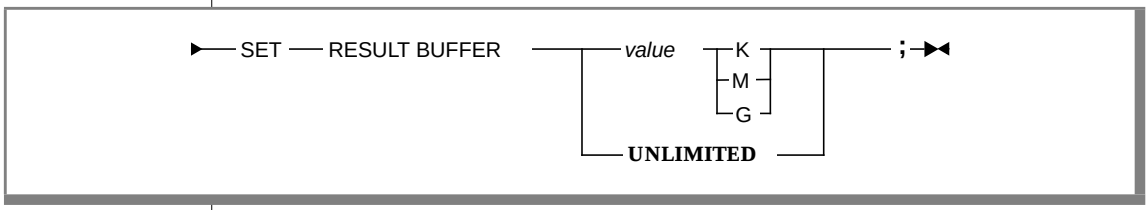
For a large result set, the read locks on the tables remain until all of the results either leave Red Brick Decision Server or are placed in the buffer. With client tools that require user input to receive more than a certain amount of data, the read locks remain on the tables until all of the results are either delivered to the client or are placed in the buffer to wait for the client.

Two SQL SET statements and two corresponding **rbw.config** TUNE parameters control the behavior of the result buffer:

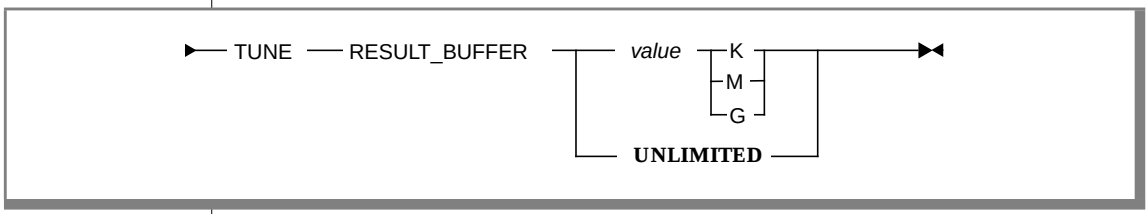
- SET RESULT BUFFER
- SET RESULT BUFFER FULL ACTION
- TUNE RESULT_BUFFER
- TUNE RESULT_BUFFER_FULL_ACTION

RESULT BUFFER Parameter

To specify the size of the buffer that holds query results until the client is ready to receive them, enter an SQL SET statement with the following syntax.



The corresponding **rbw.config** file syntax is as follows.



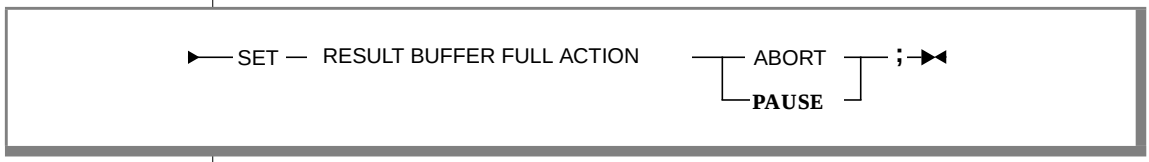
value — Specifies an integer value, which must be followed by K (kilobytes), M (megabytes), or G (gigabytes).

UNLIMITED — Indicates that there is no limit on the amount buffered. The buffer uses the same space allocated with the QUERY TEMPSPACE MAXSPILLSIZE parameter. When the RESULT BUFFER parameter is set to unlimited, the buffer size is still limited by the QUERY TEMPSPACE MAXSPILLSIZE value.

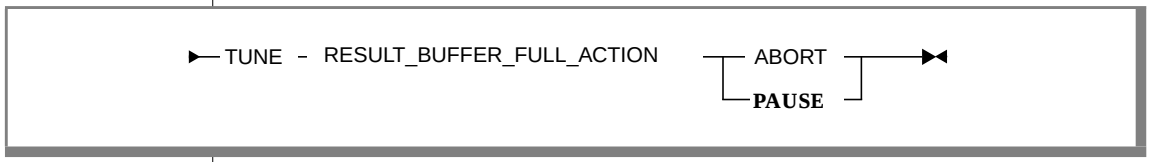
Setting a value of `0` for the RESULT BUFFER parameter specifies that no results will be buffered.

RESULT BUFFER FULL ACTION Parameter

To specify the behavior when the results buffer size specified with the SET RESULT BUFFER command is reached, enter a SET statement with the following syntax.



The corresponding **rbw.config** file syntax is as follows.



ABORT	Indicates that the query will abort when the buffer size is reached.
PAUSE	Indicates that when the buffer size is reached the query will pause until the client requests more data.

Example

The following SET statements specify a result buffer of 100 megabytes for the current session and force the query to abort when that buffer size is reached:

```
set result buffer 100M;
set result buffer full action abort;
```

Setting Segment and Partial Availability Behavior

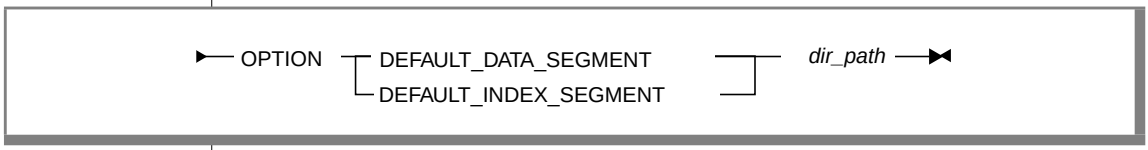
Segment creation and deletion behavior, as well as query behavior against partially available tables, are determined on a global basis by option settings in the r**bw.config** file. To override these option settings, for the current session, enter SET commands on the command line.

Location of Default Segments

You can specify a directory location for all default row data segments and another for all default index segments (that is, those segments not specifically created with a CREATE SEGMENT statement).

Syntax

To set a default directory for default data or index segments for all sessions, enter a line in the **rbw.confi**g file using the following syntax.



To set a default directory for default data or index segments for specific sessions, enter a SET command using the following syntax.



dir_path Pathname of the directory in which all default row data segments or all default index segments are to be stored.

If no default directory is specified, all default segments are stored in the database directory, as defined in the **rbw.config** file or with the **RB_PATH** environment variable.

Examples

The following examples illustrate how to specify a default location for default data and index segments.

UNIX

With **rbw.config** file entries:

```
OPTION DEFAULT_DATA_SEGMENT /dsk1/dsegs
   OPTION DEFAULT_INDEX_SEGMENT /dsk1/ixsegs
```

With SET commands:

```
set default data segment storage path 'dsk1/dsegs';
set default index segment storage path 'dsk1/ixsegs';
```

♦

WIN NT/2000

With **rbw.config** file entries:

```
OPTION DEFAULT_DATA_SEGMENT c:\dsk1\dsegs
   OPTION DEFAULT_INDEX_SEGMENT c:\dsk1\ixsegs
```

With SET commands:

```
set default data segment storage path 'c:\dsk1\dsegs';
set default index segment storage path 'c:\dsk1\ixsegs';
```

♦

Segment Drop Behavior

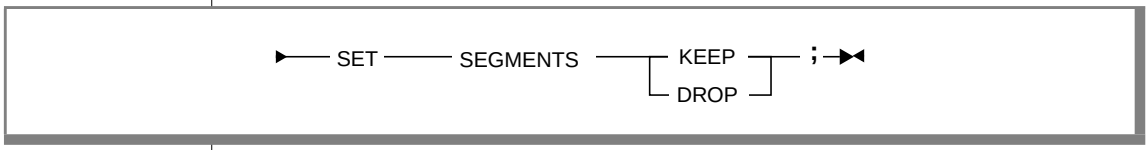
You can specify whether a user-defined segment should be dropped or kept if the table or index in that segment is dropped. (Default segments are always dropped.)

Syntax

To specify segment drop behavior for user-defined segments for all sessions, enter a line in the **rbw.config** file using the following syntax.



To specify segment- drop behavior for user-defined segments for specific sessions, enter a SET command using the following syntax.



KEEP Specifies the segment is to be kept for reuse even though the table or index it contains is dropped. The segment can be reused for another table or index. The default behavior is KEEP for user-defined segments.

DROP Specifies the segment is to be dropped if the table or index it contains is dropped.



Tip: *If the table to be dropped contains any damaged segments—default or user-defined—and the segment drop behavior is KEEP, the table cannot be dropped until the damaged segments are detached and dropped.*

Examples

Specify drop behavior for user-defined segments for all sessions with an **rbw.config** file entry, as the following example illustrates:

```
OPTION SEGMENTS DROP
```

Specify drop behavior for user-defined segments for a specific session with a SET command, as the following example illustrates:

```
set segments drop;
```

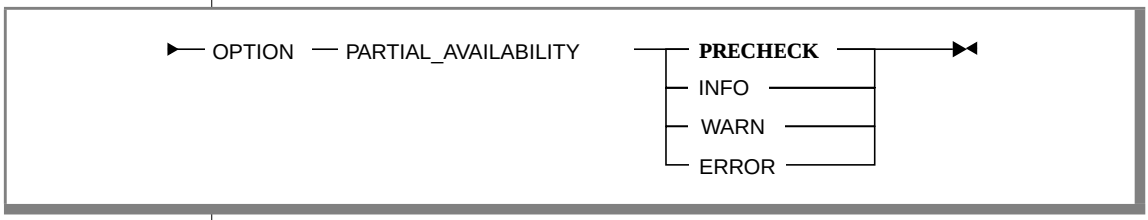
Query Behavior on Partially Available Tables

You can specify how queries behave against partially available tables. In this context, a partially available table is a table with either one or more offline row data segments or one or more offline index segments for the index to be used for that query.

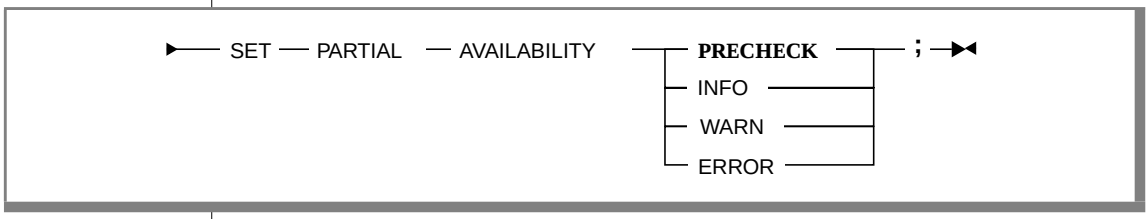
Whether indexes with offline segments are considered when the best index for a query is selected depends on the setting of the IGNORE PARTIAL INDEXES option. If a query is processed, an error or warning message is issued stating that a partial index was used, and you know that other fully available but less optimal indexes exist, you can set the IGNORE PARTIAL INDEXES option to ON to force the query processing to use the fully available index.

Syntax

To specify the query behavior for all sessions, enter a line in the **rbw.config** file using the following syntax.



To specify the query behavior for specific sessions, enter a SET command using the following syntax.



PRECHECK Specifies that table availability is to be checked before the query is processed. If a table is only partially available, an error message is issued, and the query is not processed. The default is PRECHECK.

INFO Specifies that the query is to be processed and the results returned, even if a row data or index segment is unavailable. If the results would be different if the table were fully available (that is, if the database server needs to access an offline segment to process the query), an informational message to this effect is issued along with the results.

WARN Same as INFO, but the message is a warning, not an informational message.

ERROR Specifies that the query is to be processed even if a row data or index segment is unavailable. If the results would be different if the table were fully available, no results are returned, and an error message is issued. The query might process for a significant amount of time before determining that the results might be affected.

Examples

The following examples illustrate how to specify query behavior with partially available segments.

With an **rbw.config** file entry:

```
OPTION PARTIAL_AVAILABILITY INFO
```

With a SET command:

```
set partial availability error ;
```

Use of Partially Available Indexes

You can specify whether partially available indexes should be considered when the best strategy for processing a given query is selected.

Syntax

To specify use of partially available indexes for all sessions, enter a line in the **rbw.config** file using the following syntax.



To specify use of partially available indexes for specific sessions, enter a SET command using the following syntax.



ON Specifies that only fully available indexes are to be considered in selecting the best index for a query. If no applicable index is fully available, an error message is issued, and the query fails.

OFF Specifies that all indexes, even partially available ones, are to be considered in selecting the best index for a query. If a partially available index is determined to be the best choice for a given query, the setting for the PARTIAL AVAILABILITY option controls how the query is processed.

Examples

The following examples illustrate how to specify the use of partially available indexes.

With an **rbw.config** file entry:

```
OPTION IGNORE_PARTIAL_INDEXES OFF
```

With a SET command:

```
set ignore partial indexes off ;
```

Setting the VARCHAR Column Fill Factor

The VARCHAR column fill factor is an estimate of the expected typical size of the VARCHAR column in a given table. It is specified as a percentage of the maximum column length, with a default of 10 percent, and is declared as part of a CREATE TABLE statement. To improve the performance of queries, specify an accurate fill factor.

How the Database Server Uses the VARCHAR Fill Factor

The database server uses row numbers to identify and access rows within a segment. For more information on row numbers, refer to "Pseudocolumns" on page 9-23.

The database server assigns a certain number of row numbers to each block according to the following rows-per-block formula. In a table with no VARCHAR columns, the rows are fixed length. The number of rows per block, or number of row numbers, is calculated precisely by dividing the block size (8 kilobytes) by the row size. For tables containing VARCHAR columns, the database server estimates the number of rows per block with the following formula:

```
rows per block = block size / typical row size
```

A typical row size includes the size of all the columns and uses the fill factor value for the VARCHAR column. For the effect of the VARCHAR fill factor value on the row size, refer to "Example 1: Effect of the VARCHAR Fill Factor on Number of Rows Per Block" on page 10-31.

Effect of Fill Factor on Performance

The number of row numbers reserved in each block varies inversely with the fill factor setting. A higher fill factor results in fewer row numbers per block, and a lower fill factor results in more row numbers per block.

If the fill factor is too high, too few row numbers are allowed in each block. When all of these row numbers are used, the database server writes no more rows to that block regardless of how much space is left in it. The unusable free space in each block increases the storage space required for the table and slows access because the database server must access more blocks. "Example 2: VARCHAR Fill Factor Too High" on page 10-32 illustrates this effect.

If the fill factor is too low, the number of row numbers is calculated as higher than it actually is. In this case, every block is filled, but row numbers are allocated to blocks that cannot be used because each block is too full to contain any more rows. These unused row numbers have the following effects:

- TARGET indexes are less efficient.

  Depending upon domain size, a TARGET index can have entries for every row number in the table. Unused row numbers waste space in the index, and large numbers of unused row numbers can affect performance.

  For more information on domain size, refer to the CREATE INDEX section of the *SQL Reference Guide*.

- The MAXROWS PER SEGMENT limit might be hit prematurely.

  This limit estimates the table size in the form of the maximum number of rows per segment. The unused row numbers are counted toward the maximum number of rows allowed in the segment. "Example 3: VARCHAR Fill Factor Too Low" on page 10-33 illustrates this effect.

  For more information on this parameter, refer to "Setting the MAXSEGMENTS and MAXROWS PER SEGMENT Options" on page 5-14.

A small number of unused row numbers does not significantly affect performance. Generally, it is better to have unused row numbers than unused space. Err on the side of underestimating the fill factor rather than overestimating. The best performance comes from setting the fill factor as precisely as possible.

Example 1: Effect of the VARCHAR Fill Factor on Number of Rows Per Block

This example illustrates how the VARCHAR fill factor affects the typical row size that the database server uses to estimate the number of row numbers per block.

Suppose you create a table with the following CREATE TABLE statement:

```
CREATE TABLE supplier (
   supkey integer,
   type char(20),
   name varchar (30),
   street varchar (30),
   city char (20),
   state char (5)
   zip char (10));
```

Because this statement does not specify the WITH FILLFACTOR clause, the fill factor value defaults to 10 percent for both the N**ame** and **Street** columns. The database server calculates the typical row size with this default fill factor value, as the following equation shows:

```
typical row size = length(SUPKEY) + length(TYPE) +
               ((fillfactor * length(NAME)) + 2-byte offset) +
               ((fillfactor * length(STREET)) + 2-byte offset) +
               length(CITY) + length(STATE) + length(ZIP) +
               1-byte null indicator
                  = 4 + 20 + ((10 percent * 30) + 2) +
                     ((10 percent * 30) + 2)
                  + 20 + 5 + 10 + 1
                  = 70 bytes
```

The database server calculates the number of rows per block for the **Supplier** table with the following equation:

```
rows per block = (block size - overhead) /
                      (typical row size + overhead)
               = (8192 - 4) / (70 + 2)
               = 113.72
```

Example 2: VARCHAR Fill Factor Too High

This example illustrates how too high a VARCHAR fill factor value might cause unusable free space in each block.

Suppose you create a table with the following CREATE TABLE statement that specifies a fill factor value of 90:

```
CREATE TABLE tab1 (
   col1 integer,
   col2 integer,
   col3 char (18),
   col4 varchar (100) WITH FILLFACTOR 90);
```

The database server calculates the typical row size and the number of rows per block for the **Tab1** table with the following equations:

```
typical row size = length(col1) + length(col2) + length(col3)
                   + ((fillfactor * length(col4)) +
                   2-byte offset + 1-byte null indicator
                 = 4 + 4 + 18 + (90 percent * 100) + 2 + 1
                 = 119

rows per block = (block size - overhead) /
                      (typical row size + overhead)
               = (8192 - 4) / (119 + 2)
               = 67.67
```

However, if the actual size of the values in the VARCHAR column is 70 bytes, each row requires less space. The following equations show the actual space used for this example and the amount of wasted space per block:

```
actual row size = 4 + 4 + 18 + (70) + 2 + 1
                = 99
actual space used = actual rowsize * rows-per-block
                  = 99 * 67
                  = 6633 bytes
```

The following equation estimates the amount of wasted space per block:

```
wasted space = (blocksize - overhead) - actual space used
             = 8188 - 6633
             = 1555 bytes per block
```

Example 3: VARCHAR Fill Factor Too Low

This example illustrates how too low a VARCHAR fill factor value might cause insert problems.

The **Supplier** table in "Example 1: Effect of the VARCHAR Fill Factor on Number of Rows Per Block" on page 10-31 uses the default value of 10 percent for the fill factor. The database server calculates the typical row size and the number of rows per block for the **Supplier** table with the following equations:

```
typical row size = length(SUPKEY) + length(TYPE) +
                   (fillfactor * length(NAME)) + 2-byte offset) +
                   (fillfactor * length(STREET)) + 2-byte offset) +
                   length(CITY) + length(STATE) + length(ZIP) +
                   1-byte null indicator
                 = 4 + 20 + ((10 percent * 30) + 2) +
                   ((10 percent * 30) + 2)
                   + 20 + 5 + 10 + 1
                 = 70 rows per block
                 = (block size-overhead)/(typical row
                   size+overhead)
                 = (8192 - 4) / (70 + 2)
                 = 113.72
```

However, if the actual average size of the values in the **name** column is 15 bytes and 20 bytes in the **street** column, the actual row size is 98 bytes. The block becomes full before the rows-per-block value is reached. The following equations show that the database server cannot insert 113 rows because no more space is left in the block:

```
 actual row size = 4 + 20 + (15 + 2) + (20 + 2) + 20 + 5 + 10
                 = 98 actual number of rows per block
                 = (blocksize - overhead) / (actual rowsize +
                   overhead)
                 = (8192 - 4) / (98 + 2) = 81.88
```

In this case, the database server inserts only 82 rows in the first block. The database server inserts the next row in the second block but assigns it row number 113. Row numbers 83 through 112 are unused (31 unused row numbers).

Unused row numbers count toward the value specified in MAXROWS PER SEGMENT. For example, if you specify a value of 2500 for MAXROWS PER SEGMENT and leave the default value for fill factor, you can insert only about 1800 rows, as the following equations show:

```
CREATE TABLE supplier (
   supkey integer,
   type char(20),
   name varchar (30),
   street varchar (30),
   city char (20),
   state char (5)
   zip char (10))
MAXROWS PER SEGMENT 2500;

number of blocks = MAXROWS PER SEGMENT / rows per block
                 = 2500 / 113
                 = 22.12 blocks

actual number rows insert = actual rows per block * num blocks
                          = 82 * 22
                          = 1804 rows
```

The insertion of the 1805th row returns the following error message:

```
** ERROR ** (654) Data cannot be inserted into the table because
the maximum number of rows per segment has been reached.
```

Monitoring Accuracy of the VARCHAR Fill Factor

To evaluate the effectiveness of a fill factor value, use the CHECK TABLE statement with the VERBOSE option and obtain the current fill factor value.

Using CHECK TABLE with the VERBOSE Option

The CHECK TABLE statement with the VERBOSE option displays the following pertinent segment statistics for a table.

CHECK TABLE Segment Statistics Field	Description
storage reclen	Length (in bytes) the database server expects each record to be based on the fill factor of VARCHAR columns. The typical row size formula in "Example 1: Effect of the VARCHAR Fill Factor on Number of Rows Per Block" on page 10-31 calculates this value.
average reclen	Actual average size (in bytes) of records in the column.
rows/block	Number of row numbers the database server allocates per block based on the fill factor of VARCHAR columns. "How the Database Server Uses the VARCHAR Fill Factor" on page 10-29 shows the formula to calculate this value.
unused RowIDs	Number of row numbers currently not used by actual rows.
unusable freespace	Number of bytes of free space in blocks where all assigned row numbers are used and the amount of free space exceeds the size of a typical rows. Nonzero values appear in this field when the fill factor is higher than the actual row size. Refer to "Example 2: VARCHAR Fill Factor Too High" on page 10-32.



Tip: *The* ***storage reclen*** *value and* ***average reclen*** *value should be a similar size. If you slightly underestimate the fill factor, as recommended,* ***average reclen*** *is slightly larger than* ***storage reclen****.*

The following CHECK TABLE statement produces segment statistics for the **Supplier** table. Refer to "Example 1: Effect of the VARCHAR Fill Factor on Number of Rows Per Block" on page 10-31.

```
RISQL> CHECK TABLE supplier DIRECTORY '/qa/local/sct-pubs/
   varchar-aroma/'
> VERBOSE;

INFORMATION
Table: 7  Segment: 13 is ok
No inconsistencies were detected.
```

The VERBOSE option produces segment statistics in the directory that the DIRECTORY keyword specifies. The name of the output file for the previous CHECK TABLE statement starts with the table and segment numbers, as the following filename shows:

```
chk_7_13_rep.19990909.083256.25579
```

The CHECK TABLE statement produces the following segment statistics for the **Supplier** table:

```
Segment statistics:
  active rows:9, deleted rows:0, total blocks:2, free blocks:0
  rows/block:113, storage reclen:70, average reclen:98, indirect rows:0
  max reclen:124, longest rec:106, min reclen:64, shortest rec:85
  freespace:7289, unused RowIDs:104, unusable freespace:0
```

The following descriptions explain the values in this sample output:

- The value of **storage reclen** is 70, and **average reclen** is 98, which means that the actual average length of the VARCHAR columns is longer than the fill factor. For the possible consequences of estimating too low a fill factor value, refer to "Example 3: VARCHAR Fill Factor Too Low" on page 10-33.
- The value of **freespace** is 7289, and **unused RowIDs** is 104. These unused bytes and row numbers reside mainly in the end of the last block allocated.

For more information on the syntax of the CHECK TABLE statement, refer to the *SQL Reference Guide*.

Obtaining the Current Fill Factor Value

To obtain the current fill factor value for a VARCHAR column and the lengths of all columns in a table, query the **RBW_COLUMNS** system table, as the following sample statement shows:

```
RISQL> select name, type, length, fillfactor, nulls
> from rbw_columns where tname = 'SUPPLIER';
```

This query produces the following results, which show that the current fill factor is the default value of 10 percent for the **name** and **street** columns:

```
NAME         TYPE          LENGTH FILLFA NULL

SUPKEY       INTEGER            4      0 N
TYPE         CHAR              20      0 N
NAME         VARCHAR           30     10 N
STREET       VARCHAR           30     10 N
CITY         CHAR              20      0 N
STATE        CHAR               5      0 N
ZIP          CHAR              10      0 N
```

Modifying the VARCHAR Fill Factor

You might want to adjust the fill factor for a VARCHAR column for one of the following reasons:

- To reduce the amount of wasted disk space

  For a description of how too high a fill factor value might cause wasted disk space, refer to "Example 2: VARCHAR Fill Factor Too High" on page 10-32.

- To reduce the number of row numbers per block

  For a description of how too low a fill factor value might cause TARGETjoin or INSERT problems, refer to "Example 3: VARCHAR Fill Factor Too Low" on page 10-33.

To adjust the fill factor for a VARCHAR column, use the ALTER TABLE CHANGE FILLFACTOR statement. This statement does not take effect until you execute an ALTER TABLE statement that adds or drops a column. Then the whole table is rewritten using the new fill factor.

For example, you might want to adjust the fill factor to reduce the amount of wasted space in the **Tab1** table in "Example 2: VARCHAR Fill Factor Too High" on page 10-32. The **Tab1** table currently has a fill factor value of 90 percent for the **Col4** VARCHAR column, but the actual average length is 70 bytes, which is 70 percent of the VARCHAR(100) specification in the CREATE TABLE statement.

To more closely match the actual average length of the VARCHAR column values, specify a fill factor of 70. The following sample ALTER TABLE statement changes the fill factor for the **Tab1** table:

```
ALTER TABLE tab1 ALTER COLUMN col4
   CHANGE FILLFACTOR 70;
```

For syntax information on the ALTER TABLE statements, refer to the *SQL Reference Guide*.

Setting the Index Fill Factor

The index fill factor is used to determine how full to fill each new node of a B-TREE index when the node is initially built. (Each node in a B-TREE index corresponds to a file system block.) If new index nodes are not completely filled, subsequent incremental load operations can insert entries in the index without causing the nodes (blocks) to split. Such splits slow incremental loads. Indexes built with a fill factor of less than 100 percent require more storage but can provide better incremental load, update, and insert performance.

The default fill factor is 100 percent. If the table and index are to be loaded once and then used only for query operations, with no incremental load, insert, or update operations, 100 percent is the appropriate fill factor. However, if you know that the table will grow, you can specify a fill factor that will permit the index nodes to accommodate the growth without splitting.

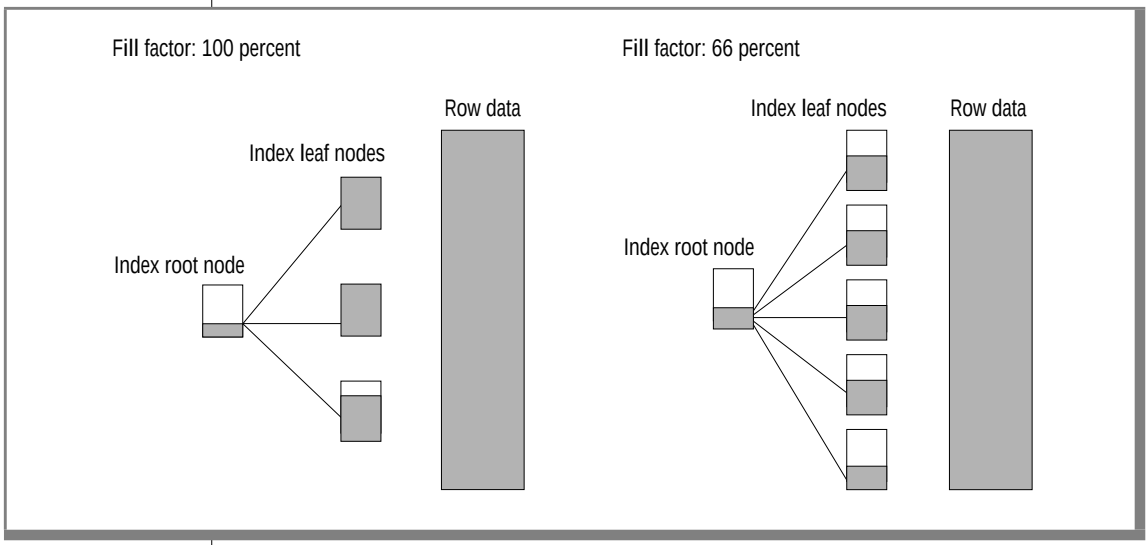
You can specify system default fill factors in the **rbw.config** file. To specify fill factors for individual indexes, use the CREATE INDEX or ALTER INDEX statements. All fill factors default to 100 percent unless otherwise specified. To force the TMU to use a user-defined fill factor, you must include the OPTIMIZE clause of the LOAD DATA statement. If the OPTIMIZE clause is not present, the TMU uses 100-percent fill factors, ignoring any specified fill factors.

For more information about fill factor in relation to index size estimates, refer to "Index Fill Factors" on page 4-23.

Example

The following figure illustrates an index with a fill factor of 100 percent and the same index with a fill factor of 66 percent, after an initial load.

Figure 10-3
Fill Factor and Index Nodes

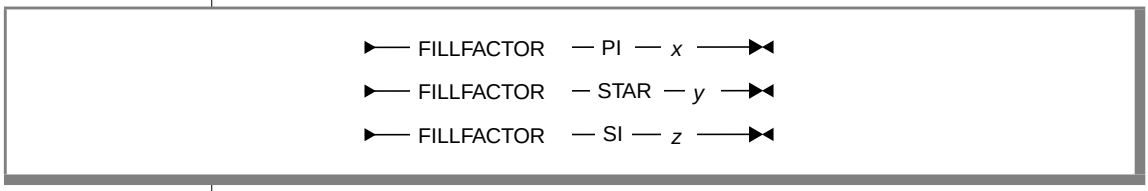


At 100 percent, each node fills completely before a new node is started. All the index data fits in three leaf nodes (blocks), two of which are completely full and a third almost full. If more rows of data are added that require an index entry in either of the full nodes, the nodes must be split.

At 66 percent, the same index requires almost five nodes (blocks), but space remains in each index node to accommodate additional row data added to the table.

Syntax

To specify system default fill factors for all sessions, enter the appropriate lines in the **rbw.config** file using the following syntax.



FILLFACTOR PI *x*	Specifies the system default fill factor for all primary indexes.
FILLFACTOR STAR *y*	Specifies the system default fill factor for all STAR indexes.
FILLFACTOR SI *z*	Specifies the system default fill factor for all non-STAR secondary indexes (any non-STAR index created with a CREATE INDEX statement).
x, *y*, *z*	Integers ranging from 1 to 100, indicating percentage. Default values are `100`.

Usage Notes

To specify a different fill factor for a specific user-created index, use a CREATE INDEX…WITH FILLFACTOR *x* statement when you create the index.

To specify or change a fill factor for a specific automatically created index, use an ALTER INDEX…CHANGE FILLFACTOR *x* statement after determining the index name from the **RBW_INDEXES** table.

When a new index is created, the fill factor used for that index is determined from the **rbw.config** file for automatic indexes and from the CREATE INDEX statement or **rbw.config** file for user-defined indexes. If no fill factor is specified, `100` is used. This fill factor is stored in the **RBW_INDEXES** system table. For each new node added to that index during the initial or incremental load operations, the fill factor used is the one stored in **RBW_INDEXES**. To change this fill factor on an index-by-index basis, use the ALTER INDEX statement.

For those indexes built automatically by the TMU (primary key indexes and any B-TREE, STAR, and TARGET indexes created with a CREATE INDEX statement prior to the load operation), the OPTIMIZE clause must be present in the LOAD DATA statement in order to use user-specified fill factors.

Finding the Fill Factor Used for a Specific Index

To determine the fill factor used for a specific index, query the **RBW_INDEXES** system table as follows:

```
select name, fillfactor
from rbw_indexes
where name = 'index_name';
```



Important: *Actual use of the fill factor depends on whether the TMU OPTIMIZE mode is set.*

Deciding Whether to Change Default Fill Factors

To decide whether to change a default fill factor (100 percent for all indexes), determine whether the tables in your database are expected to grow over time. If any of them will grow, estimate the expected growth and decide if the reduction in load time is enough to justify the additional space required by filling nodes less than 100 percent full.



Important: *This discussion assumes the expected growth throughout the index is uniform. If you expect growth at the ends of the index or in new segments of a segmented index, use the default fill factor.*

To see how the fill factor affects the amount of storage required for an index, consider the following formula, used to calculate how many elements are stored in each index node:

$$\text{Elements per index node} = \left\lfloor \frac{fillfactor}{100} \times \frac{8172}{keysize + 6} \right\rfloor$$

Each index node is 8172 bytes and corresponds to one 8-kilobyte file system block minus 20 bytes overhead, and *keysize* is the width of the key column, plus 6 bytes of address.

Therefore, a smaller fill factor reduces the maximum size allowed for an index key.

Example

Assume a key size of 10 bytes on a table with 50,000 rows. To calculate the number of 8-kilobyte blocks required to store the index for various fill factors, use the formulas in "Estimating the Size of Indexes" on page 4-22. The results are as follows.

Fill Factor	Elements Per Node	Blocks
10	81	627
50	409	124
100	817	63

If this table is to be loaded once with no anticipated additions, use a fill factor of 100.

If you expect this table to grow to twice its initial size, a fill factor of 50 is a reasonable choice. Likewise, if the table is loaded initially with data that is only 10 percent of what you expect it to contain, a fill factor of 10 is reasonable.



Important: *If you choose a low fill factor, the resulting indexes can be large and affect query performance negatively.*

Changing an Index Fill Factor

The fill factor specifies the percentage of space filled in new index nodes as they are created. You might want to change the fill factor for an index for one of the following reasons:

- To reduce the amount of wasted disk space

  The amount of empty space left in each node affects the amount of space required when an index is built.

- To improve incremental load, update, and insert performance

  Fill factor affects the frequency with which nodes fill completely and must be split.

To change the fill factor used during the creation of new index nodes, use the ALTER INDEX…CHANGE FILLFACTOR statement. Altering the fill factor on an index that is not empty affects only new nodes built during future load operations in optimize mode. It does not change or rebuild existing nodes. To cause existing nodes to be rebuilt with a new fill factor, you must change the fill factor and then REORG the index or drop and re-create it.

Creating Additional Indexes

When you create a table containing a primary key, a B-TREE index is automatically created on the primary key columns. Creating additional indexes often improves query performance. In determining whether to create additional indexes, weigh the improvement in performance against the additional space required to store the index and the time required to build and update the index when changes are made to the tables upon which the index is based.

For fact tables that are queried with constraints on foreign key columns, you can improve query performance by creating STAR indexes using the foreign keys that will be constrained. The order of the foreign keys in a STAR index affects performance on particular queries. The best performance is gained from a STAR index whose leading foreign key matches the query constraint.

If all the columns in the primary key are also columns in foreign keys, creating a STAR index that includes all the primary key columns might make the primary key B-TREE index redundant. In this case, you can save disk space and processing overhead by dropping the primary key B-TREE index after creating the primary key STAR index.

You can also improve performance for multitable joins where the tables are related by primary key / foreign key relationships by enabling TARGETjoin processing. To enable TARGETjoin, create indexes on the foreign key columns of the referencing (fact) table. For information on TARGETjoin processing, refer to "TARGETjoin Query Processing" on page 10-61 and to Chapter 4, "Planning a Database Implementation."

If some columns contain weakly selective constraints, consider creating TARGET indexes to greatly improve query performance for queries constraining on those columns. For detailed information, refer to "TARGET Indexes" on page 4-10.

Whenever a dimension table contains a foreign key that is not also a primary key (that is, the table references an outboard table), consider either creating an additional B-TREE or TARGET index on each foreign key column for better performance or creating a STAR index.

The EXPLAIN statement can help you determine what indexes are being used for a given query. For information about EXPLAIN, refer to "EXPLAIN Statement" on page 10-57.

For more information about using additional indexes to improve performance and their requirements, refer to "Determining When to Create Additional Indexes" on page 4-4.

Understanding Query Processing

Red Brick Decision Server evaluates a query and automatically decides the best way to process it, based on the indexes that are available. The database server goes through many phases while processing a query. The following sections list the join algorithms Red Brick Decision Server uses, illustrate the phases of query processing, and discuss how to use the EXPLAIN command. This command shows you what phases a particular query goes through and what steps you can take to optimize query performance.

Join Algorithms

Generally, queries that involve joins might require tuning. Red Brick Decision Server uses the following algorithms to process joins:

- STARjoin
- TARGETjoin
- B-TREE one-to-one match (nested loop join)
- Hybrid hash join
- Naive one-to-one match (cross join)

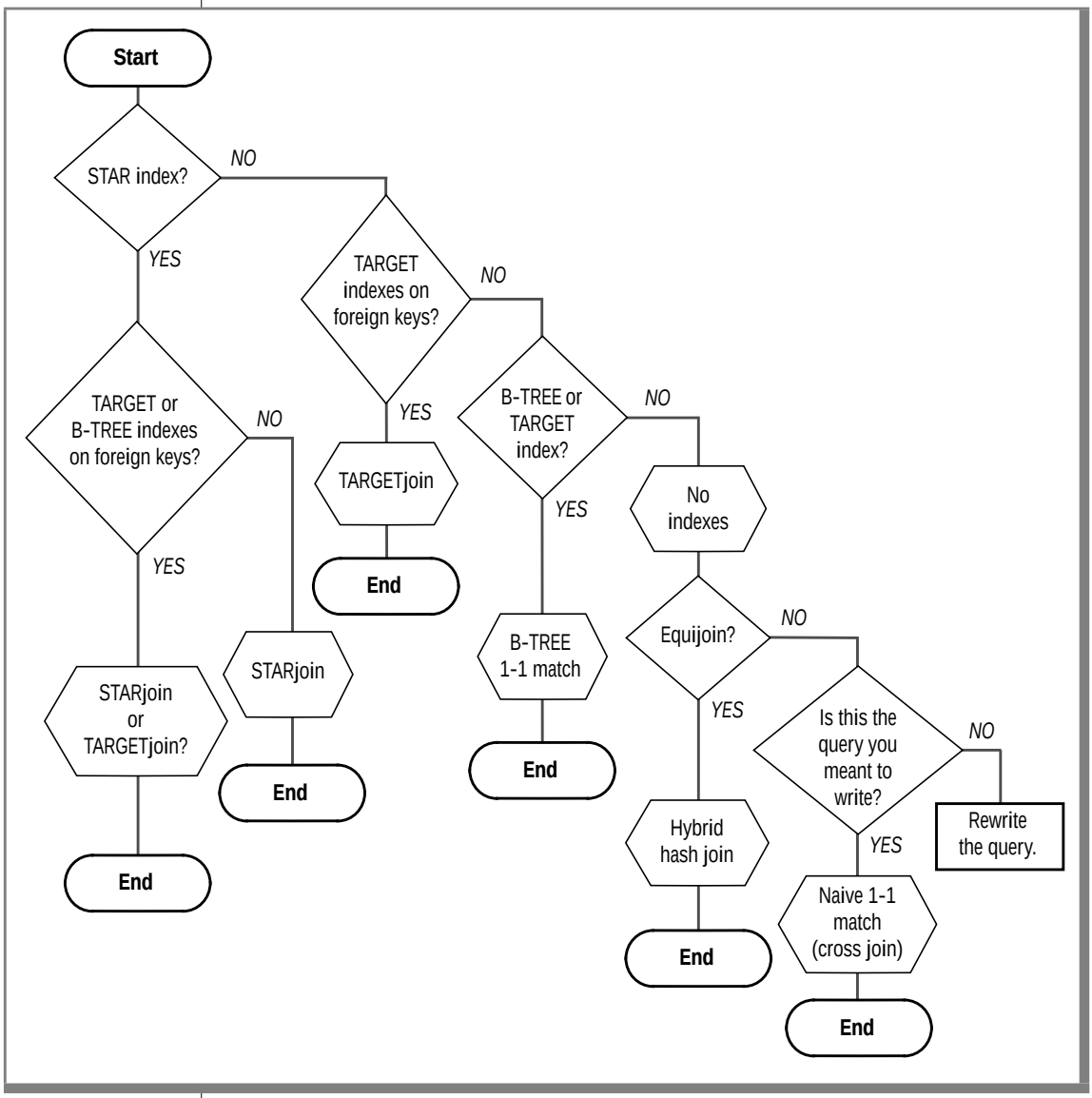
The database server chooses the join method based on which indexes are available. At runtime, the database server evaluates the query and makes decisions about the query execution plan based on the following criteria:

1. If the appropriate STAR index exists to join the tables, the query uses the STARjoin algorithm. An appropriate STAR index contains some or all of the keys that are constrained in the query.
2. If the appropriate STAR index exists to join the tables, and if TARGET or B-TREE indexes exist on the foreign key columns in the referencing (fact) table, the query uses either the STARjoin or TARGETjoin algorithm, depending on which has the best indexes available for the join operation.
3. If the appropriate STAR index does not exist, but TARGET indexes exist on the foreign key columns in the referencing (fact) table, the query uses the TARGETjoin algorithm.
4. If the appropriate STAR index does not exist, but either a B-TREE or a TARGET index is present over the joining columns, that index is used with the B-TREE one-to-one match algorithm. If both a B-TREE and a TARGET index exist, the database server chooses the best index for the join operation.
5. If no indexes are present over the joining columns and the join is an equijoin (the query constraints are equality conditions), the hybrid hash join algorithm is used.
6. If no indexes are present over the joining columns and the join is *not* an equijoin, the cross join algorithm is used. The cross join algorithm calculates all possible combinations of the joining columns (the Cartesian product). Therefore, cross joins are disallowed unless the OPTION CROSS JOIN parameter is set to ON. This requirement ensures that users do not issue cross join queries inadvertently; by omitting a join condition, for example.

These criteria are simplified for the purpose of this discussion. Many other variables add complexity to the choices Red Brick Decision Server makes.

The following figure illustrates the decision-making process that the database server uses to join tables.

Figure 10-4
Decisions to Join Tables



Operator Model

Red Brick Decision Server uses an object-oriented operator model to evaluate a query. Stages of the query are broken down into *operators*, which are elemental pieces of the query execution plan. Each operator has its own portion of work to do before passing the execution along to the next operator.

The output of the EXPLAIN statement lists the operators that are used in a given query execution plan. By understanding what each operator does and which indexes are being used in a query, you can devise ways of improving query performance by one or more of the following means:

- Rewriting the query
- Adding or dropping indexes
- Changing the schema design
- Changing the values of tuning parameters
- Modifying the physical layout of your database and database files

For more information on the EXPLAIN statement, refer to "EXPLAIN Statement" on page 10-57.

The following alphabetical list of the operators provides a brief description of each, as well as a brief description of the fields associated with each operator that appears in the EXPLAIN output.

Advisor

This operator produces Advisor information for Informix Vista.

B-TREE 1-1 Match

Given a key value, this operator looks it up in the index.

Join Type	Lists the type of join performed by the operator. Possible values: `InnerJoin`, `LeftOuterJoin`, `RightOuterJoin`, `FullOuterJoin` For information on the different types of joins, refer to the *SQL Reference Guide*.
Index	Lists the name of the index used in the operation and the name of the table on which the index is defined.

B-TREE Scan

This operator scans an index or a range of keys in an index.

Reverse Order	For multiple-column indexes, indicates if the index is being scanned in the key order (`FALSE`) of the index or in the reverse key order (`TRUE`) of the index. It is more efficient to scan an index in the order of its keys than in the reverse order. If Reverse Order = T, you might improve performance by creating an additional index with the key order reversed in the CREATE INDEX statement. Possible values: `TRUE`, `FALSE`
Predicate	A predicate is a restricted condition on the query. For example: `...where promo_type = 400` Predicates can have one condition or multiple conditions.

Start-Stop Predicate

A start-stop predicate is a predicate with a restrictive range, such as a conjunction in the WHERE clause. This approach allows only a restricted portion of the index to be scanned, which increases the efficiency of the index scan, thus improving query performance. For example, the following predicate restricts the range on **Col1** between 5 and 8:

```
... where col1 > 5 and col1 < 8
```

If the domain of **Col1** is all integers and an index is defined on **Col1**, this predicate allows a very efficient, restricted scan of the index, which is good for query performance.



Tip: *Red Brick Decision Server uses start-stop predicates whenever possible, even if the WHERE clause contains no conjunctions.*

Bit Vector Sort

This operator sorts the RBW_SEGID and RBW_ROWNUM bits from the pseudocolumns containing that information.

Check

This operator performs consistency checking of tables and indexes.

Choose Plan

This operator represents a dynamic decision point made at runtime. Only one of the choices is executed at runtime.

Num Prelims	Lists the number of preliminary operations that are performed before the choice is made as to which plan to use. These preliminary operations are performed regardless of which plan is selected.
Num Choices	Lists the number of choices the database server chooses from at runtime.
Type	Lists the type of plan. Possible values: `Unknown`, `General`, `STARjoin`, `Sequence`, `Prerun`.

Delete

This operator performs the DELETE operation.

Delete All	Indicates (FALSE) if there is a search condition on the DELETE operation, as specified in the WHERE clause of the DELETE statement, and (TRUE) if there is no search condition. Possible values: TRUE, FALSE
Table	The name of the table from which the rows are deleted.
Constraint Name	The name of the constraint that references another table involved in the DELETE operation.

Delete Cascade

This operator finds the data to be deleted in the referenced table for a DELETE CASCADE operation.

Delete All	Indicates (FALSE) if there is a search condition on the DELETE operation, as specified in the WHERE clause of the DELETE statement, and (TRUE) if there is no search condition. Possible values: TRUE, FALSE
Table	The name of the table from which the rows are deleted.

Delete Refcheck

This operator checks whether the DELETE operation will violate referential integrity. If it finds a violation of referential integrity, the DELETE operation is disallowed.

Delete All	Indicates (FALSE) if there is a search condition on the DELETE operation, as specified in the WHERE clause of the DELETE statement, and (TRUE) if there is no search condition. Possible values: TRUE, FALSE
Table	The name of the table from which the rows are deleted.
Constraint Name	The name of the constraint that references another table involved in the DELETE operation, used to check referential integrity.

Exchange

This operator splits an operation for parallelism.

Exchange Type	Lists the type of operation to be parallelized. Possible values: `Unknown`, `Functional Join`, `STAR-join`, `Table Scan`, `Upper Hash 1-1 Match`, `Lower Hash 1-1 Match`.

Execute

This operator coordinates the interaction between operators. Execute is the first operator in an EXPLAIN report and handles the database server side disposition of the data.

For client / server connectivity, this operator formats and packs up the data for shipping through ODBC to the client. It sets the table lock type.

Table Locks	Possible values: `Read_Only`, `Read_Key`, `Read_Data`, `Write_Data`, `Write_Blocking`.

For Export operations, this operator formats and writes the data to the output file or pipe.

Functional Join

Given the segment ID and row number, this operator reads the row.

Number of Tables	The number of tables involved in the operation and their names.

General Purpose

This operator is used for dynamic substitutions such as SELECT COUNT(*).

Operation — Describes the operation to be performed by the operator.

- Count: Optimization for COUNT(*) processing based on table size when there is no predicate, grouping, HAVING clause, and so on.
- Textsize: Processing of `'SELECT @@textsize'` variable selection, used by some query tools.

Hash 1-1 Match

This operator performs a hybrid hash join.

Join Type — Lists the type of join performed by the operator.

Possible values: `InnerJoin`, `LeftOuterJoin`, `RightOuter-Join`, `FullOuterJoin`, `RightSemiJoin`, `RightAntiSemiJoin`

For information on the different types of joins, refer to the *SQL Reference Guide*.

Hash AVL Aggregate

This operator is used for aggregate and GROUP BY processing.

Grouping — Indicates if GROUP BY processing is used.

Possible values: `TRUE`, `FALSE`

Distinct — Indicates if SELECT DISTINCT processing is used.

Possible values: `TRUE`, `FALSE`

Insert

This operator is used to insert into a table.

Table — The name of the table being inserted into.

Mode — Indicates whether the INSERT operation is written to disk as soon as each row is received (`Immediate`) or whether the data is stored in a buffer and then written to disk all at once when the operation has completed (`Delayed`).

`Values` indicates an INSERT operation using the VALUES keyword, as follows:

```
insert into table1 values (a, b);
```

Possible values: `Values`, `Delayed`, `Immediate`

Merge Sort

This operator is used to perform sorts.

Distinct — Indicates if SELECT DISTINCT processing is used.

Possible values: `TRUE`, `FALSE`

Naive 1-1 Match

This operator is used to compute the Cartesian product (cross join) of two tables.

RISQL Calculate

This operator is used to process RISQL display functions.

Sample Scan

This operator returns a sample of its input.

Simple Merge

This operator takes two input lists and combines them.

Sort 1-1 Match

This operator performs a matching sort of two sorted lists.

Match type	Possible values: `Union`, `Intersect`, `Except`

STARjoin

This operator performs the STARjoin processing.

Join Type	Lists the type of join performed by the operator. Possible value: `InnerJoin` For information on the different types of joins, refer to the *SQL Reference Guide*.
Num Facts	Lists the number of fact (referencing) tables involved in the operation.
Num Potential Dimensions	Lists the number of potential dimension (referenced) tables involved in the operation.
Fact Table	Lists the names of the fact tables involved in the operation.
Potential STAR Indexes	For each fact table, lists the names of the potential STAR indexes involved in the STARjoin operation.
Dimension Tables	Lists the names of the dimension (referenced) tables that participate in the STAR index.

Subquery

This operator is used to process a subquery.

Scalar	Indicates a scalar subquery. Possible values: TRUE, FALSE
Correlated	Indicates a correlated subquery. Possible values: TRUE, FALSE

Table Scan

This operator scans a table.

Table	Indicates the name of the table being scanned.
Predicate	Indicates the predicate (the restrictive condition) on the table being scanned.

TARGETjoin

This operator performs the TARGETjoin processing, which efficiently joins tables related by primary key / foreign key relationships.

Table	Indicates the name of the table involved in the operation.
Predicate	Indicates the predicate (the restrictive condition) on the operation.
Num Indexes	Indicates the number of indexes involved in the operation.
Index(s)	Indicates the name of each index involved in the operation.

TARGET Scan

This operator performs TARGET index processing and processing of certain INTERSECT and UNION operations that use a B-TREE index.

Table	Indicates the name of the table involved in the operation.
Predicate	Indicates the predicate (the restrictive condition) on the operation.
Num Indexes	Indicates the number of indexes involved in the operation.
Index(s)	Indicates the name of each index involved in the operation.

Update

This operator performs an update operation.

Mode	Indicates whether the UPDATE operation is written to disk as soon as each row is processed (`Immediate`) or whether the data is stored in a buffer and then written to disk all at once when the operation completes (`Delayed`). Possible values: `Delayed`, `Immediate`
Table	Indicates the name of the table being updated.

Virtual Table Scan

This operator manages internal results stored in virtual memory.

EXPLAIN Statement

The EXPLAIN statement provides output detailing a query execution path. EXPLAIN indicates which operators and indexes are used. Use EXPLAIN to understand how a query is going to execute. Based on the information, you might decide to create or drop indexes to tune the query for better performance.

Some parts of the query execution path are determined dynamically during query execution. In those cases, EXPLAIN indicates the possible paths. When you run a query with SET STATS INFO enabled, the actual query execution path for each Choose Plan operator is printed in informational messages. You can also access graphical EXPLAIN support from the ISQL window of the Administrator tool. This tool displays a graphical view of the query operator tree and includes additional details on operator arguments.

EXPLAIN also shows where parallelism is used in the part of the report on the Exchange operator. When you run a query with SET STATS INFO enabled, the degree of parallelism that the query actually used for each Exchange operator is printed in informational messages.

In the EXPLAIN report, each operator is indicated by dashes (—) followed by its name in capital letters. Each operator name is followed by an ID number that is unique to the query. This ID is used to track which operator is performing which part of the query-processing work. Descriptive information about the actions the operator is performing follows the ID number. For information on the operators, refer to "Operator Model" on page 10-47.

The following example shows the report on the Execute operator from a typical EXPLAIN report:

```
- EXECUTE (ID: 0) 2 Table locks (table, type): (PROMOTION,
Read), (SALES, Read_Only)
```

Each EXPLAIN query-processing report begins with the word `EXPLANATION` followed by the Execute operator.

Example

The following example shows the output from EXPLAIN for a simple join of the **Sales** and **Promotion** tables from the **Aroma** database:

```
RISQL> explain select sales.promokey, dollars
>from promotion, sales
>where sales.promokey = promotion.promokey;

EXPLANATION
[
- EXECUTE (ID: 0) 2 Table locks (table, type): (PROMOTION,
Read_Only), (SALES, Read_Only)
--- EXCHANGE (ID: 1) Exchange type: Table Scan
----- TABLE SCAN (ID: 2) Table: SALES, Predicate: <none>
]
RISQL>
```

In this example, the report shows read-only locks on the **Promotion** and **Sales** tables. The tables will be joined using a B-TREE one-to-one match join with the **Promotion_pk_idx** B-TREE index.

Example

The following example adds a constraint to the previous example:

```
RISQL> explain select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;

EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PROMOTION,
Read_Only), (SALES, Read_Only), (PERIOD, Read_Only), (PRODUCT,
Read_Only), (STORE, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 2; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
      -- TABLE SCAN (ID: 3) Table: PROMOTION, Predicate:
(PROMOTION.PROMO_TYPE)
= (400)
    }

    Choice: 1; Choose Plan [id : 1] {
    EXCHANGE (ID: 4) Exchange type: Functional Join
    -- FUNCTIONAL JOIN (ID: 5) 1 tables: SALES
    ---- EXCHANGE (ID: 6) Exchange type: STARjoin
      ------ STARJOIN (ID: 7) Join type: InnerJoin, Num facts: 1,
Num potential
dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}

    Choice: 2; Choose Plan [id : 1] {
     EXCHANGE (ID: 8) Exchange type: Table Scan
    -- FUNCTIONAL JOIN (ID: 9) 1 tables: PROMOTION
     ---- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
[Table: PROM
OTION, Index: PROMOTION_PK_IDX]
   ------ TABLE SCAN (ID: 11) Table: SALES, Predicate: <none>
}

]
RISQL>
```

In addition to the read-only locks from the previous example, this plan shows a choice that the database server will make at runtime. Choice 1 is a STARjoin using the STAR index **Sales_star_idx**, and choice 2 is a B-TREE one-to-one match (nested loop join) using the **Promotion_pk_idx** B-TREE index. Parallelism might occur in several places (shown by the Exchange operators). In Choice 1, the STARjoin can be parallelized. In Choice 2, the B-TREE one-to-one match can be parallelized.

If you now run the query with SET STATS INFO enabled, you get the following results:

```
RISQL> set stats info;
RISQL> select sales.promokey, dollars
>      from promotion, sales
>      where sales.promokey = promotion.promokey
>      and promotion.promo_type = 400;

** STATISTICS ** (500) Compilation = 00:00:00.19 cp time,
00:00:00.23 time, Logical IO count=57
PROMOKEY    DOLLARS
        172    348.00
        172    128.00
...
        181     48.00
        165     79.75
** STATISTICS ** (1457) EXCHANGE (ID: 4) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 6) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 1) STARjoin on 1 tables
was done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 1) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:06.48 cp time, 00:00:06.99
time, Logical IO count=307
** INFORMATION ** (256) 392 rows returned.
RISQL>
```

This output shows that Choice 1, the STARjoin, was actually executed. The degree of parallelism on the Exchange (`ID=4`) was 4, and the degree of parallelism on Exchange (`ID=6`) was 1.

TARGETjoin Query Processing

Red Brick Decision Server includes a family of join methods, one of which is the TARGETjoin bit-mapped join. TARGETjoin processing works on star schemas or any schema that has primary key / foreign key relationships and is a join method complementary to STARjoin technology. It uses TARGET indexes on the foreign keys of a fact table (B-TREE indexes on multicolumn foreign keys) to join the table to the tables referenced by the foreign keys. This section explains how TARGETjoin processing works and provides information on how to use and administer a database to take advantage of this join method. The following topics are included:

- How to Use TARGETjoin Processing
- When to Use TARGETjoin Processing
- Examples
- Reading EXPLAIN Output for a TARGETjoin Query
- Summary and Recommendations

How to Use TARGETjoin Processing

This section explains how to enable TARGETjoin processing and under what conditions Red Brick Decision Server uses TARGETjoin processing to run a query.

Create TARGET or B-TREE Indexes on Foreign Keys of Fact Table

To enable TARGETjoin processing, you create indexes (TARGET or B-TREE) on the foreign keys of the *fact* table. For single-column foreign keys, create TARGET indexes on the foreign key column. For multicolumn foreign keys, create B-TREE indexes on the concatenation of the foreign key columns. The reason for creating B-TREE indexes on multicolumn foreign keys is that TARGET indexes are single-column indexes and therefore cannot be created on a multicolumn foreign key. After you create the indexes and run queries, TARGETjoin processing is selected automatically when it is the best choice.

For the purpose of describing the TARGETjoin functionality, consider a fact table to be any table that references other tables by foreign key references. This definition includes fact tables in a simple star schema that reference dimension tables by foreign key references, and it also includes dimension tables that reference outboard tables by foreign key references.

Rules for TARGETjoin Query Processing

To understand how Red Brick Decision Server chooses whether to use TARGETjoin processing, consider the following cases:

- One or more STAR index is present.
- No STAR index is present.

When the Fact Table Has at Least One STAR Index

If at least one STAR index covering any of the keys is constrained in a query, a TARGETjoin query plan is generated based on the STARjoin query plan if the following conditions are satisfied:

- At least one STAR index exists covering some or all of the constrained keys of the query.
- TARGET or B-TREE indexes exist on all the single-column fact table foreign keys that are both constrained in the query and keys of the qualifying STAR indexes.
- B-TREE indexes exist on all the multicolumn fact table foreign keys that are both constrained in the query and keys of the qualifying STAR indexes.

If these conditions are not satisfied, a query cannot use TARGETjoin processing, but the query completes using another join method. When these conditions are satisfied, the optimizer considers both STARjoin and TARGETjoin processing and automatically chooses the best index and join method for each query.

When the Fact Table Has No Qualifying STAR Indexes

If no STAR index covering the keys is constrained in a query, a TARGETjoin query plan is generated when the following conditions are satisfied:

- No STAR index exists covering any of the constrained keys of the query.
- The query joins two or more dimension tables to the fact table.
- At least two of the constrained dimensions have TARGET indexes on the corresponding fact table foreign keys (B-TREE index on multi-column foreign keys).



Important: *Dimensions that reference a single-column foreign key must have a TARGET index on that foreign key, or they are not considered for a TARGETjoin query plan, even if a B-TREE index exists on the foreign key.*

If these conditions are not satisfied, the query uses another join method. When these conditions are satisfied, the query plan generated does not include STARjoin as a possible join method. It includes a choice of a table scan or TARGETjoin processing. You can use the EXPLAIN statement to see whether TARGETjoin processing is an option for a particular query. For examples of EXPLAIN output that show TARGETjoin query plans, refer to "Reading EXPLAIN Output for a TARGETjoin Query" on page 10-69.

Turning Off TARGETjoin Query Processing

If you do not want query plans generated that include TARGETjoin processing, make sure either that no indexes exist on the foreign key columns of the fact table(s) or that the segments containing the indexes are in the OFFLINE state. You can drop the indexes with the DROP INDEX statement or you can bring the indexes to the OFFLINE state with the ALTER SEGMENT...OFFLINE statement. For information on these statements, refer to the *SQL Reference Guide*.

When to Use TARGETjoin Processing

When deciding whether to use TARGETjoin processing, consider the following questions:

- Do your queries perform well already?
- Is your schema appropriate for TARGETjoin processing?
- What are the trade-offs between creating more STAR indexes and creating indexes on the foreign key columns?

Evaluate Query Performance

If your query performance is already good, do not create the indexes to enable TARGETjoin processing. The sole purpose is to speed the performance of queries. If that performance is already good, there is no reason to incur the administrative cost of creating and maintaining more indexes.

However, the question “Is performance good?” is often not so easily answered. Do all queries need to complete in less than 10 seconds, or can some take 10 minutes? Is it acceptable for some queries that are not issued often to take several hours?

Only you and your user community can answer these questions. The users always appreciate faster queries. Consider if absolute performance is more important than longer downtime during database maintenance, for example. Evaluate your query performance and decide whether the costs of creating the indexes to enable TARGETjoin processing are worth the benefits.

For more information about the costs associated with TARGETjoin processing, refer to “Administration Considerations for TARGETjoin Processing” on page 4-14.

Schema Types

TARGETjoin processing joins tables that have primary key / foreign key relationships. These relationships occur in a wide variety of schemas, most notably in star schemas. But many star schemas expand on the simple star schema and have outboard tables. Such schemas are sometimes referred to as *snowflake* schemas. The following figure shows such a schema.

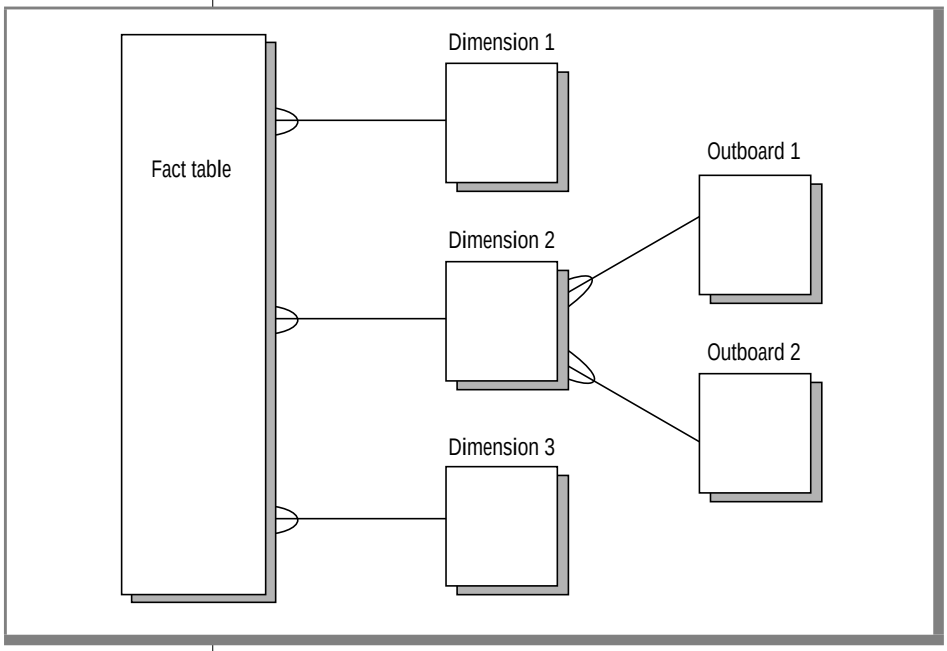


Figure 10-5
Snowflake Schema

With a snowflake schema, TARGETjoin processing is possible to join the dimension tables to the fact table and to join the outboard tables to the dimension tables that reference them. To enable TARGETjoin processing in the preceding figure, you must create TARGET indexes on the foreign key columns of the Fact table and on the foreign key columns of the Dimension 2 table.

Many STAR Indexes Versus TARGETjoin Processing

Creating foreign key indexes to enable TARGETjoin processing is an alternative to creating many STAR indexes. For schemas with many dimensions, there are fewer TARGET indexes to create than there are potential STAR indexes. For example, a schema with one fact table and ten dimension tables has ten factorial (10!) or 3,628,800 possible STAR indexes. The same schema has ten possible foreign key indexes.

When the number of dimensions in a schema is relatively small, the number of potential STAR indexes is much lower, and it is much easier to create a set of STAR indexes that perform well for a large variety of queries. For example, on a schema with one fact table and four dimension tables, two or three STAR indexes can provide excellent performance for virtually any query against the database. As the number of dimensions increases, however, it becomes more difficult to cover a large variety of queries with a relatively small number of STAR indexes.

If you know exactly what queries will be run against the database, you can always create STAR indexes that are well suited for those queries. As the number of STAR indexes grows, however, this strategy becomes more and more impractical. Exactly what *large* is depends on what is practical in your unique situation. A good compromise between ultimate performance and manageability is to create a few STAR indexes and to also create foreign key indexes to enable TARGETjoin processing. This approach will provide excellent performance on some queries and good performance on all queries.

Examples

The examples in this section are based on a modified version of the **Aroma** database, the sample database shipped with Red Brick Decision Server. To create this database, create indexes on the foreign keys of the **Sales** table of the **Aroma** database using the following CREATE INDEX statements:

```
create target index sales_perkey_target_idx on sales (perkey);
create target index sales_promokey_target_idx on sales(promokey);
create target index sales_storekey_small_target_idx
     on sales (storekey) domain small;
create index sales_classkey_prodkey_btree_idx
     on sales (classkey, prodkey);
```



Tip: *The TARGET index on the foreign key that references the* ***Store*** *table is created as DOMAIN SMALL because there are only 18 unique values. The index on the foreign key that references the* ***Product*** *table is a B-TREE index because it is a multi-column foreign key.*

Query That Chooses TARGETjoin

Suppose you want to answer the following business questions:

- What are the dollar values of the sales on each promotion of type 900 in Atlanta?
- What are the total sales for each promotion and for all the promotions of type 900 in Atlanta?

The following query against the modified **Aroma** database answers this question and is executed using TARGETjoin processing:

```
RISQL> set stats info;
RISQL> select substr(promo_desc, 1, 20) as PROMO_DESC,
>         substr(store_name, 1, 25) as STORE_NAME, dollars
> from sales natural join store natural join promotion
> where city like 'Atlanta%' and promo_type = 900
> order by 1
> break by 1 summing 3
> ;

** STATISTICS ** (500) Compilation = 00:00:00.41 cp time,
00:00:00.20 time, Logical IO count=77
PROMO_DESC           STORE_NAME                       DOLLARS
Christmas special    Olympic Coffee Company            210.00
Christmas special    NULL                              210.00
Easter special       Olympic Coffee Company            420.00
Easter special       Olympic Coffee Company             30.00
Easter special       Olympic Coffee Company            150.00
Easter special       NULL                              600.00
NULL                 NULL                              810.00
** STATISTICS ** (1457) EXCHANGE (ID: 19) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 26) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 3) Choice: 3 was chosen 1
times.
** STATISTICS ** (1461) CHOOSE PLAN (ID: 3) TARGETjoin was done 1
times.
** STATISTICS ** (500) Time = 00:00:00.20 cp time, 00:00:00.74
time, Logical IO count=348
** INFORMATION ** (256) 7 rows returned.
RISQL>
```

This query constrains on the **Store** table (`city like 'Atlanta%'`) and on the **Promotion** table (`promo_type = 900`). These two tables are referenced by foreign keys from the **Sales** table. The STAR index on the **Sales** table was created with the following CREATE INDEX statement:

```
create star index sales_star_idx
     on sales (sales_date_fkc, sales_product_fkc,
               sales_store_fkc, sales_promo_fkc);
```

Notice the order of the keys in the STAR index. The foreign key constraints that reference the **Store** and **Promotion** tables are the last two keys of this index. A STARjoin query performs best when the leading keys of the index are constrained in the query. Adding a constraint on the **Period** table to the previous query causes it to use STARjoin processing to join the tables, as follows:

```
RISQL> set stats info;
RISQL> select substr(promo_desc, 1, 20) as PROMO_DESC,
>           substr(store_name, 1, 25) as STORE_NAME, dollars
> from sales natural join store natural join promotion natural
join period
> where city like 'Atlanta%'
> and promo_type = 900
> and year = 1999
> order by 1
> break by 1 summing 3
> ;

** STATISTICS ** (500) Compilation = 00:00:00.17 cp time,
00:00:00.31 time, Logical IO count=135
PROMO_DESC           STORE_NAME                          DOLLARS
Easter special       Olympic Coffee Company                30.00
Easter special       Olympic Coffee Company               150.00
Easter special       NULL                                 180.00
NULL                 NULL                                 180.00
** STATISTICS ** (1457) EXCHANGE (ID: 10) Parallelism over 1 times
High: 3 Low: 3 Average: 3.
** STATISTICS ** (1457) EXCHANGE (ID: 14) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 3) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 3) STARjoin on 1 tables
was done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 3) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:02.08 cp time, 00:00:02.41
time, Logical IO count=318
** INFORMATION ** (256) 4 rows returned.
RISQL>
```

This query chooses STARjoin processing instead of TARGETjoin processing because it constrains on the **Period** table. The **sales_date_fkc** constraint from the **Period** table is the leading key of the STAR index. Therefore, the STAR index offers optimal performance in this case.

Reading EXPLAIN Output for a TARGETjoin Query

When TARGETjoin processing is a possible choice for a query execution path, the Choose Plan operator in the EXPLAIN output shows as many as three choices:

- Table scan (might include B-TREE 1-1 match, depending on the indexes available)
- STARjoin
- TARGETjoin

The join method is chosen at runtime and can be displayed with the statistics messages issued when SET STATS INFO is enabled.

STAR and TARGET Plan

The following example of the EXPLAIN output for a query has STARjoin processing, table scan, and TARGETjoin processing as execution options. This query is run against the modified **Aroma** database (with TARGET indexes).

```
RISQL> explain select count(*)
> from sales natural join period natural join store
> where year = 1999
> and store_name like 'C%';

** STATISTICS ** (500) Compilation = 00:00:00.24 cp time,
00:00:00.23 time, Logical IO count=75
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PERIOD,
Read_Only), (STORE, Rea
d_Only), (SALES, Read_Only), (PRODUCT, Read_Only), (PROMOTION,
Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 2; Num choices: 3; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
     -- TABLE SCAN (ID: 3) Table: PERIOD, Predicate: (PERIOD.YEAR)
= (1999)
     }

    Prelim: 2; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 4)
      -- TABLE SCAN (ID: 5) Table: STORE, Predicate:
((STORE.STORE_NAME) =< ('Cÿ
ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) && ((STORE.STORE_NAME) >= ('C') )
      }

Choice: 1; Choose Plan [id : 1] {
       HASH AVL AGGR (ID: 6) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
    -- EXCHANGE (ID: 7) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 8) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
    ------ EXCHANGE (ID: 9) Exchange type: STARjoin
     -------- STARJOIN (ID: 10) Join type: InnerJoin, Num facts:
1, Num potential dimensions: 4, Fact Table: SALES, Potential
Indexes: SALES_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}

    Choice: 2; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 11) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct : FALSE;
    -- EXCHANGE (ID: 12) Exchange type: Table Scan
     ---- HASH AVL AGGR (ID: 13) Log Advisor Info: FALSE,
```

```
Grouping: FALSE, Distinct: FALSE;
    ------ FUNCTIONAL JOIN (ID: 14) 1 tables: PERIOD
      -------- BTREE 1-1 MATCH (ID: 15) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
    ---------- FUNCTIONAL JOIN (ID: 16) 1 tables: STORE
      ------------ BTREE 1-1 MATCH (ID: 17) Join type: InnerJoin;
Index(s): [Table: STORE, Index: STORE_PK_IDX]
      -------------- TABLE SCAN (ID: 18) Table: SALES, Predicate:
<none>
}

    Choice: 3; Choose Plan [id : 1] {
       HASH AVL AGGR (ID: 19) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
    -- EXCHANGE (ID: 20) Exchange type: Functional Join
      ---- HASH AVL AGGR (ID: 21) Log Advisor Info: FALSE,
 Grouping: FALSE, Distinct: FALSE;
    ------ EXCHANGE (ID: 22) Exchange type: TARGETjoin
      -------- TARGET JOIN (ID: 23) Table: SALES, Predicate: <none>
; Num indexes: 2 Index(s): Index: SALES_PERKEY_TARGET_IDX ,Index:
SALES_STOREKEY_SMALL_TARGET_IDX
    ---------- FUNCTIONAL JOIN (ID: 24) 1 tables: PERIOD
     ------------ VIRTAB SCAN (ID: 25)
    ---------- FUNCTIONAL JOIN (ID: 26) 1 tables: STORE
     ------------ VIRTAB SCAN (ID: 27)
}

]
** STATISTICS ** (500) Time = 00:00:00.02 cp time, 00:00:00.10
time, Logical IO count=0
** INFORMATION ** (256) 60 rows returned.
RISQL>
```

The Choose Plan operator provides three choices: STARjoin, Table Scan with B-TREE 1-1 match, and TARGETjoin.

TARGET Only Plan

The following example illustrates the EXPLAIN output for a query that does not have a STARjoin execution option. The query is the same as in the previous example, but this query is run against an **Aroma** database with TARGET indexes on the **Sales** table foreign keys and no STAR indexes.

Informix does not recommend dropping your STAR indexes when using TARGETjoin processing. This example illustrates only what the EXPLAIN output looks like in situations where there are no STAR indexes.

```
RISQL> explain select count(*)
> from sales natural join period natural join store
> where year = 1999
> and store_name like 'C%';

** STATISTICS ** (500) Compilation = 00:00:00.23 cp time,
00:00:00.23 time, Logical IO count=75
EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PERIOD,
Read_Only), (STORE, Read_Only), (SALES, Read_Only), (PRODUCT,
Read_Only), (PROMOTION, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 2; Num choices: 3; Type:
StarJoin;

    Prelim: 1; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 2)
     -- TABLE SCAN (ID: 3) Table: PERIOD, Predicate: (PERIOD.YEAR)
= (1999)
     }

    Prelim: 2; Choose Plan [id : 1] {
     BIT VECTOR SORT (ID: 4)
     -- TABLE SCAN (ID: 5) Table: STORE, Predicate:
((STORE.STORE_NAME) =< ('Cÿ ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ') ) &&
((STORE.STORE_NAME) >= (
'C') )
     }

Choice: 1; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 6) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
   -- EXCHANGE (ID: 7) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 8) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct: FALSE;
   ------ EXCHANGE (ID: 9) Exchange type: STARjoin
     -------- STARJOIN (ID: 10) Join type: InnerJoin, Num facts:
1, Num potential dimensions: 4, Fact Table: SALES, Potential
Indexes: SALES_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION
}

    Choice: 2; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 11) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct : FALSE;
   -- EXCHANGE (ID: 12) Exchange type: Table Scan
     ---- HASH AVL AGGR (ID: 13) Log Advisor Info: FALSE,
Grouping: FALSE, Distinct: FALSE;
   ------ FUNCTIONAL JOIN (ID: 14) 1 tables: PERIOD
```

```
     -------- BTREE 1-1 MATCH (ID: 15) Join type: InnerJoin;
Index(s): [Table: PERIOD, Index: PERIOD_PK_IDX]
   ---------- FUNCTIONAL JOIN (ID: 16) 1 tables: STORE
     ------------ BTREE 1-1 MATCH (ID: 17) Join type: InnerJoin;
Index(s): [Table: STORE, Index: STORE_PK_IDX]
     -------------- TABLE SCAN (ID: 18) Table: SALES, Predicate:
<none>
}

    Choice: 3; Choose Plan [id : 1] {
      HASH AVL AGGR (ID: 19) Log Advisor Info: FALSE, Grouping:
FALSE, Distinct : FALSE;
   -- EXCHANGE (ID: 20) Exchange type: Functional Join
     ---- HASH AVL AGGR (ID: 21) Log Advisor Info: FALSE,
Grouping: FALSE, Distinct: FALSE;
   ------ EXCHANGE (ID: 22) Exchange type: TARGETjoin
     -------- TARGET JOIN (ID: 23) Table: SALES, Predicate: <none>
; Num indexes: 2 Index(s): Index: SALES_PERKEY_TARGET_IDX ,Index:
SALES_STOREKEY_SMALL_TARGET_IDX
   ---------- FUNCTIONAL JOIN (ID: 24) 1 tables: PERIOD
    ------------ VIRTAB SCAN (ID: 25)
   ---------- FUNCTIONAL JOIN (ID: 26) 1 tables: STORE
    ------------ VIRTAB SCAN (ID: 27)
}

]
** STATISTICS ** (500) Time = 00:00:00.04 cp time, 00:00:00.09
time, Logical IO count=0
** INFORMATION ** (256) 60 rows returned.
RISQL>
```

This query has two choices of join methods: table scan and TARGETjoin. If you run this query with SET STATS INFO enabled, you can see that it runs using TARGETjoin, which is Choice 2 in the EXPLAIN output.

```
RISQL> select count(*)
> from sales natural join period natural join store
> where year = 1999
> and store_name like 'C%';

** STATISTICS ** (500) Compilation = 00:00:00.40 cp time,
00:00:00.40 time, Logical IO count=75

       4561
** STATISTICS ** (1457) EXCHANGE (ID: 7) Parallelism over 1 times
High: 4 Low: 4 Average: 4.
** STATISTICS ** (1457) EXCHANGE (ID: 9) Parallelism over 1 times
High: 1 Low: 1 Average: 1.
** STATISTICS ** (1458) CHOOSE PLAN (ID: 1) Choice: 1 was chosen 1
times.
** STATISTICS ** (1459) CHOOSE PLAN (ID: 1) STARjoin on 1 tables
was done 1 times.
** STATISTICS ** (1460) CHOOSE PLAN (ID: 1) used Index
SALES_STAR_IDX of Table SALES 1 times for STARjoin.
** STATISTICS ** (500) Time = 00:00:03.86 cp time, 00:00:04.15
time, Logical IO count=194
** INFORMATION ** (256) 1 rows returned.
RISQL>
```

To learn which TARGET indexes were used in the query, see the names of the indexes in the EXPLAIN output of the query, shown in the previous example.

Summary and Recommendations

TARGETjoin processing is a complementary technology to STARjoin processing. It works well when an optimal STAR index is not available. The following diagram shows a schema that is an ideal candidate for TARGETjoin processing to complement STARjoin processing.

This simple star schema has a single fact table and ten dimension tables referenced by foreign key / primary key relationships.

Such a schema can be found in a large variety of applications. For example, it could be a retail schema where the fact table is a **Sales** table with dimensions such as **Period**, **Product**, **Market**, **Customer**, and so on. It could also be a health insurance claims database where the fact table is a **Claims** table with dimensions such as **Member**, **Provider**, **Occupation**, **Physician**, and so on.

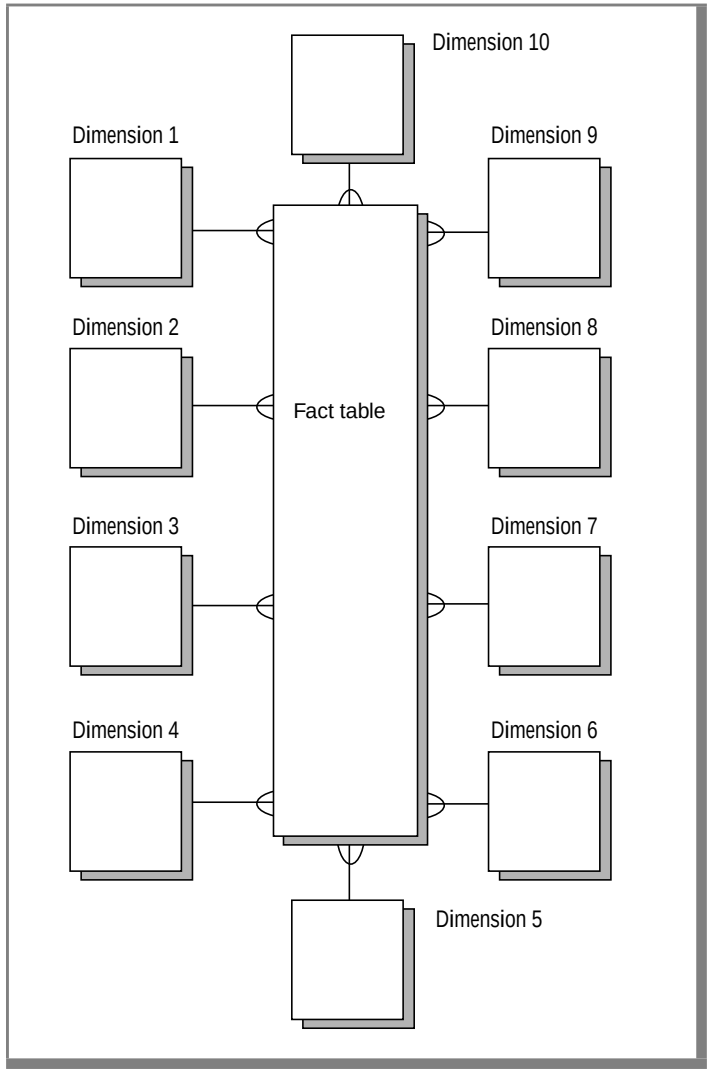


Figure 10-6 Schema Candidate for TARGETjoin Processing

With this type of schema—one with a large number of dimension tables—you might need to create many STAR indexes in order for STARjoin processing to work well over a large variety of ad-hoc queries.

With TARGETjoin processing, instead of creating many STAR indexes, you can create one or two STAR indexes and then create TARGET indexes on the foreign keys of the fact table.

Indexes to Create

For this example, assume you know that about 70 percent of your queries constrain on dimensions 1 through 4, and the rest are ad hoc queries that constrain on different combinations of all ten dimensions. In this case, a good strategy is to create the following indexes:

- One STAR index covering all of the dimensions, with the leading key the one the fact table is segmented on (for example, the time dimension)
- A second STAR index covering dimensions 1 through 4, which are used in 70 percent of your queries
- You might also want the leading key of this STAR index to be the time dimension, or whatever dimension the fact table is segmented on
- Ten TARGET indexes, one on each of the fact table foreign keys

The STAR index that covers all of the dimension table keys has two main purposes:

- It provides good query performance over a large variety of queries.
- It enables STARjoin / TARGETjoin query plans to be generated for any query against this database, as described in "When the Fact Table Has at Least One STAR Index" on page 10-62.

This strategy gives optimal performance on the core 70 percent queries, and it also gives good performance on virtually any ad hoc query that users might make.

Large Dimension Table

Suppose further that Dimension 10 is a large table named **Customer** with 1,000,000 rows. In this situation, TARGETjoin processing might not offer the best performance on queries that loosely constrain on the large **Customer** table. But these queries should perform well with STARjoin processing, particularly if the **Customer** dimension is the last key in the STAR index and other keys in that STAR index are constrained as well. Therefore, the **Customer** table (or whatever your large table is) should be the last key in the STAR index that covers all your dimensions.

Experiment

All schemas and data are different. Determining the best index implementation requires some experimentation and evaluation of your query performance. If query performance is already good with one or two STAR indexes, you probably do not need to add any additional indexes.

This example is a simplified case. Your situation obviously is different. But if you refer to these examples as guidelines and if your schema is a good candidate for TARGETjoin processing, it will complement the STARjoin processing that works well in 70 percent of queries and greatly improve performance for your database server.

Estimates of Fact Table Selectivity

Because the Red Brick server estimates the percentage of fact table rows constrained by a query in order to choose the optimal plan (a STARjoin or TARGETjoin plan versus a table scan plan), it is important to understand how these estimates are calculated, especially with respect to dimension tables that are *preloaded* with data that is not referenced in the fact table.

A typical preloaded dimension is the time dimension. Administrators often find it convenient to load time dimensions with dates that run a number of years into the future. As the obsolete years of fact table data roll off, the referenced rows in the dimensions are left intact. Therefore, the time dimension contains old rows that are no longer referenced by the fact table as well as future rows that are not yet referenced.

It follows that a query constraint on a preloaded dimension might result in a skewed estimate of the fact table selectivity. For example, say the constraint on the time dimension is for a specific year:

```
where year = 2000
```

If the time dimension contains dates from January 1, 1995, through December 31, 2004, the selectivity *on the dimension* is about 10 percent (1 out of 10 years). However, the selectivity on the referencing fact table that records actual sales history for the same period is much larger. If the most current load into the fact table occurred on September 30, 2000, and all years prior to 1999 have rolled off, the selectivity on the fact table would be over 40 percent (3 quarters out of 7).

To estimate fact table selectivity as accurately as possible with respect to these kinds of constraints, the optimizer attempts to establish the number of rows from the dimension that are physically *referenced* by the fact table. This optimization is possible when the dimension table:

- Is the leading dimension in a STAR index on the fact table
- Is not a common dimension in a query that requires a multifact join

The estimate is based on the range of fact table rows that could possibly be constrained by the query, regardless of the actual constraints in the query. How accurate the estimate is depends on whether the referenced rows are physically contiguous in the dimension table. If they are not contiguous, it will not be as accurate.

If either of the criteria is not met, the server calculates the selectivity by using the row count for the whole dimension table. *Therefore, avoid preloading a dimension table unless a STAR index exists with that dimension as its leading key.*

The following diagram illustrates the complete formula for estimating selectivity:

$$\text{Estimated rows} = \frac{\langle \text{count(*) from fact\_table} \rangle \ \langle \Pi \text{rows selected from dimension tables} \rangle}{\Pi \langle \text{row counts for dimension tables} \rangle}$$

The *row counts for dimension tables* in this formula is calculated for each dimension, either using a referenced row count or a total row count, as discussed earlier in this section. The same formula applies to the calculation of parallel join tasks, as discussed in Chapter 11, "Tuning a Warehouse for Parallel Query Processing."

Example: Selectivity Estimates for Preloaded Dimensions

This example illustrates how an accurate selectivity estimate is calculated when a preloaded dimension is the leading dimension of a STAR index.

Scenario

This example is based on the following scenario:

- A database contains the last seven quarters of actual sales history, including the current quarter.
- The **Time** dimension is referenced by the **Sales** fact table.
- The **Time** dimension contains 10 years of data—a total row count of 3,652 rows (1995 through 2004).
- The **Sales** table contains approximately 1,000,000 rows, including sales figures from Q1 1999, through Q3 2000.
- A STAR index exists on the fact table, with the **timekey** foreign key as the leading key of the index:

```
create star index star1 on sales (timekey,...,...)
```

- The following query is executed:

```
select product_name, month, sum(dollars) as monthly_sales
from product, time, sales
where product.productkey = sales.productkey
   and time.timekey = sales.timekey
   and year = 2000
group by product_name, month;
```

Selectivity Estimate

The query has a selectivity of approximately 7 percent on the dimension table (270 rows) but probably about 40 percent on the fact table. Because the STAR index is present, with the foreign key of the preloaded dimension as its leading key, the selectivity is estimated as follows:

$$\text{Estimated rows} = \frac{(1,000,000) \times 270}{638} = 423,197$$

$$\text{Selectivity} = \frac{423,197}{1,000,000} = 42.3 \text{ percent}$$

The referenced row count covers the year 1999 and the first three quarters of 2000 (approximately 638 days). In turn, over 400,000 fact table rows are expected to be constrained. The selectivity estimate of 42.3 percent would result in the choice of a table scan plan, which would be the better choice in this case. In most cases, STARjoin plans are faster when 30 percent or less of the fact table rows are constrained.

If the same database did not have a STAR index with the time dimension as its leading key, the selectivity would have to be based on the row count of the dimension (3652 rows), and the result would be the choice of a suboptimal STARjoin:

$$\text{Estimated rows} = \frac{(1000,000,000) \times 270}{3652} = 73,932$$

$$\text{Selectivity} = \frac{73,932}{1,000,000} = 7.39 \text{ percent}$$

Using Synonyms to Control Fact-to-Fact Joins

You can use synonyms to perform a hash join or a B-TREE 1-1 match join on queries that would normally use STARjoin processing. This approach is particularly useful with fact-to-fact joins in complex schemas when the fact-to-fact STARjoin operation is not performing well.

To use STARjoin processing on a multiple fact table join, each fact table must have at least one foreign key reference to a common dimension. Additionally, the fact tables must have STAR indexes whose shared foreign keys are in the same relative order.

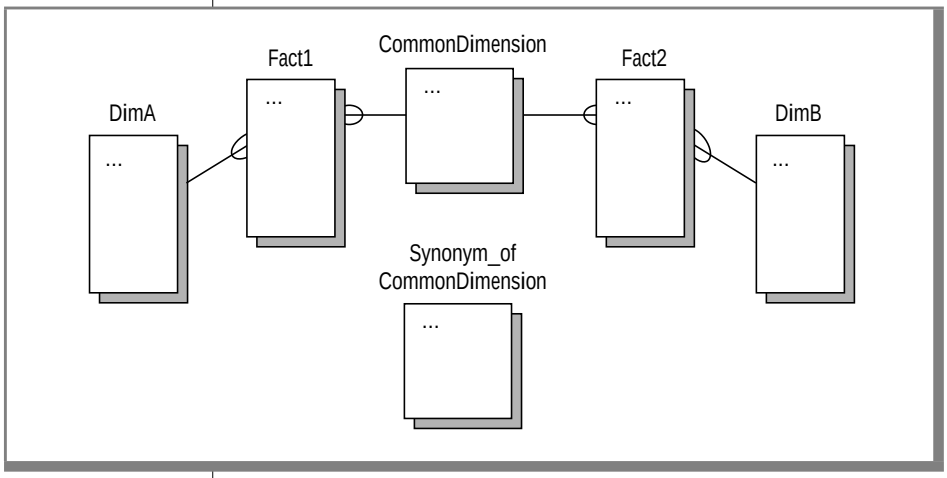


Figure 10-7 Multiple Fact Table Join

In the multiple fact table schema in the preceding figure, assume indexes exist with the following definitions:

```
create star index STAR_FACT1
     on fact1(CommonDimensionKey, DimAKey) ;
create star index STAR_FACT2
     on fact2(CommonDimensionKey, DimBKey) ;
```

The following query uses the STAR indexes for the fact-to-fact join operation:

```
select dimA.column, fact1.column, fact2.column, dimB.column
from dimA, fact1, CommonDimension, fact2, dimB
where DimA.column = <value1>
  and DimB.column = <value2>
  and DimA.DimAKey = fact1.DimAKey
  and fact1.CommonDimensionKey =
      CommonDimension.CommonDimensionKey
  and fact2.CommonDimensionKey =
      CommonDimension.CommonDimensionKey
  and fact2.DimBKey = DimB.DimBKey ;
```

If this fact-to-fact STARjoin query is not performing well, you can use the ALTER TABLE...ALTER CONSTRAINT statement to move a foreign key constraint to reference a synonym instead of the base table to which the synonym refers. Any STAR indexes that reference the base table are not considered in the query plan when the synonym (instead of the base table) is specified in the query. Also, any STAR indexes that reference the synonym are not considered when the base table is specified in the query. Red Brick Decision Server can consider a different set of indexes depending on whether the base table or the synonym is included in the query.

The following ALTER TABLE...ALTER CONSTRAINT statement changes the constraint on one of the fact tables to reference the synonym instead of the **CommonDimension** table:

```
alter table fact1 alter constraint col1
     references synonym_of_commondimension;
```

The foreign key constraint from table **Fact1** now points to the synonym instead the **CommonDimension** table, as the following figure shows.

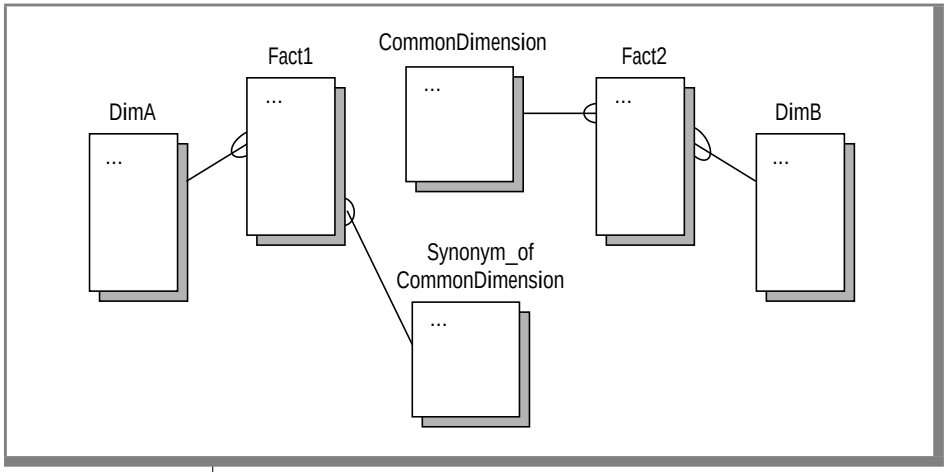


Figure 10-8 Multiple Fact Table Join Using Synonym Table

Now if you specify the synonym instead of the **CommonDimension** table in your query, a fact-to-fact STARjoin does not occur.

```
select dimA.column, fact1.column, fact2.column, dimB.column
from dimA, fact1, Synonym_of_CommonDimension, fact2, dimB
where DimA.column = <value1>
  and DimB.column = <value2>
  and DimA.DimAKey = fact1.DimAKey
  and fact1.CommonDimensionKey =
     Synonym_of_CommonDimension.CommonDimensionKey
  and fact2.CommonDimensionKey =
     Synonym_of_CommonDimension.CommonDimensionKey
  and fact2.DimBKey = DimB.DimBKey ;
```

This query uses a HASH join to join to the results of two single-fact table STARjoin operations instead of the multiple fact table STARjoin. For some queries, this approach might improve performance. However, performance depends on many factors, including schema definition, available indexes, how tightly constrained the query is, uniformity of the data, and so on. The use of synonyms provides an additional tuning tool for multiple fact table joins, but a certain amount of experimentation is required to ascertain whether it will enhance performance on your database.

Tip: *The synonym and the base table share the same physical data. They only differ logically. Query results are the same selecting from a synonym or from the table to which the synonym refers.*

For the syntax of the ALTER TABLE and CREATE SYNONYM statements, refer to the *SQL Reference Guide*.

Making SQL-Based Improvements

In some cases, changing the way queries are constructed results in improved performance. Determining when and how to make these changes requires an in-depth knowledge of SQL, but the following examples provide some limited suggestions. You can also find some ideas about alternative methods of accomplishing tasks in the *SQL Self-Study Guide* and the *SQL Reference Guide*.

UNION Versus Interdimensional ORs

A query that uses the union of two or more queries can potentially run faster than a query that constrains the referenced (dimension) tables with an OR operator. The results returned are the same.

Example

The following SQL fragments illustrate a case where performance can be improved by splitting a query with OR constraints on referenced tables into two queries whose results are combined with a UNION operation. Tables **T2** and **T3** are independent referenced tables and reflect different dimensions of the data. That is, the two tables do not reference each other.

Interdimensional OR:

```
select ... from t1,t2,t3
   where t2.col1 = 'x'
         or
         t3.col1 = 'y'
```

UNION:

```
select ... from t1, t2
   where t2.col1 = 'x'
union
select ... from t1, t3
   where t3.col1 = 'y'
```

Subquery in the FROM Clause Versus Correlated Subquery

A query that uses a subquery in the FROM clause often runs faster than an equivalent query (a query that answers the same question and returns the same results) with a correlated subquery in the SELECT list. Therefore, re-writing a correlated subquery as a subquery in the FROM clause might provide a substantial performance improvement.

Example

The following query, from the **Aroma** database, contains a correlated subquery in the select list:

```
RISQL> select outer_product.prod_name as aroma_product,
         sum(outer_sales.dollars) as dollars_jan_98,
         (select  sum(inner_sales.dollars)
           from sales as inner_sales,
           product as inner_product, period as inner_period
           where inner_sales.prodkey = inner_product.prodkey
             and inner_sales.classkey = inner_product.classkey
             and inner_sales.perkey = inner_period.perkey
             and inner_period.year = outer_period.year + 1
             and inner_period.month = outer_period.month
             and inner_product.prod_name =
             outer_product.prod_name
             and inner_product.pkg_type like 'No pkg%'
             ) as dollars_jan_99
       from sales as outer_sales, product as outer_product,
          period as outer_period
       where  outer_sales.prodkey = outer_product.prodkey
          and outer_sales.classkey = outer_product.classkey
          and outer_sales.perkey = outer_period.perkey
          and outer_period.year = 1998
          and outer_period.month = 'JAN'
          and outer_product.pkg_type like 'No pkg%'
       group by outer_product.prod_name, outer_period.year,
          outer_period.month
       order by outer_product.prod_name;

AROMA_PRODUCT                DOLLARS_JAN_98  DOLLARS_JAN_99
Aroma Roma                         10460.75        15055.00
Aroma baseball cap                  1113.00         1049.40
...

** STATISTICS ** (500) Time = 00:00:06.44 cp time, 00:00:06.25
time, Logical IO count=4445
** INFORMATION ** (256) 25 rows returned.
RISQL>
```

The following query, from the **Aroma** database, uses a subquery in the FROM clause:

```
RISQL> select aroma_product, dollars_jan_98, dollars_jan_99
       from
             (select product.prod_name, sum(dollars)
               from sales, product, period
               where sales.prodkey = product.prodkey
                  and sales.classkey = product.classkey
                  and sales.perkey = period.perkey
                  and period.year = 1998
                  and period.month = 'JAN'
                  and product.pkg_type like 'No pkg%'
               group by product.prod_name, period.year,
                  period.month )
              as sales_alias1 (aroma_product, dollars_jan_98)
           natural join
             (select product.prod_name, sum(dollars)
               from sales, product, period
               where sales.prodkey = product.prodkey
                  and sales.classkey = product.classkey
                  and sales.perkey = period.perkey
                  and period.year = 1999
                  and period.month = 'JAN'
                  and product.pkg_type like 'No pkg%'
               group by product.prod_name, period.year,
                  period.month )
              as sales_alias2 (aroma_product, dollars_jan_99)
       order by aroma_product;

AROMA_PRODUCT                 DOLLARS_JAN_98  DOLLARS_JAN_99
Aroma Roma                          10460.75        15055.00
Aroma baseball cap                   1113.00         1049.40
...

** STATISTICS ** (500) Time = 00:00:03.30 cp time, 00:00:03.87
time, Logical IO count=380
** INFORMATION ** (256) 25 rows returned.
```

This query completes in 3.87 seconds, while the previous query (with the correlated subquery) completes in 6.25 seconds. Both queries return the same results. The actual difference in the response time of a correlated subquery versus a subquery in the FROM clause differs depending on the query, the data, and other system-dependent factors.



Tip: *Actual query response times vary from system to system.*

Chapter

11

Tuning a Warehouse for Parallel Query Processing

Parallel Query Tuning Parameters 11-4

Enabling Parallel Query Processing 11-5

Limiting I/O Contention with the FILE_GROUP Parameter. 11-6

Allowing Parallelism Within Disk Groups with the GROUP Parameter 11-8

Limiting Available Tasks 11-10
- TOTALQUERYPROCS 11-11
- QUERYPROCS 11-11

Setting Minimum Row Requirements with ROWS_PER_TASK Parameters 11-13
- ROWS_PER_SCAN_TASK 11-14
- ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK 11-17
  - Estimated Rows 11-18
  - Number of Tasks 11-18
  - Enabling Parallelism for a STARjoin 11-19
  - Number of Rows in Dimension Tables 11-19

Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters 11-23
- FORCE_SCAN_TASKS 11-26
- FORCE_FETCH_TASKS and FORCE_JOIN_TASKS 11-27
- FORCE_HASHJOIN_TASKS 11-29

Enabling Partitioned Parallelism for Aggregation 11-31

System Considerations for Parallel Tasks 11-32

Analysis of System Resources and Workload 11-34
Disk Usage 11-34
For Data 11-35
For STAR Indexes 11-35
Memory Usage 11-36
CPU Allocation 11-37

Tuning for Specific Query Types 11-38
Parallel STARjoin Queries 11-38
Density 11-39
Number of Parallel Tasks 11-39
Mix of Parallel Tasks and File Groups 11-39
Considerations for Multiuser Environments 11-40
Parallel Table Scans 11-41
SuperScan Technology 11-42
About Reasonable Values 11-42

Basic Guidelines 11-43

In This Chapter

You can improve query performance by using multiple tasks to process queries against large tables. Because multiple tasks consume more system resources—CPU, disk, processes, and memory—than a single task, several parameters in the **rbw.config** file allow you to control the amount of parallel processing, balancing query response time against the available system resources and demands of other users. Queries that involve a relation scan of a large table or a STAR index are candidates for parallel processing.

This chapter describes query performance improvements that parallel query processing can provide. This chapter covers the following topics:

- Parallel Query Tuning Parameters
- Enabling Parallel Query Processing
- Limiting I/O Contention with the FILE_GROUP Parameter
- Allowing Parallelism Within Disk Groups with the GROUP Parameter
- Limiting Available Tasks
- Setting Minimum Row Requirements with ROWS_PER_TASK Parameters
- Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters
- Enabling Partitioned Parallelism for Aggregation
- System Considerations for Parallel Tasks
- Analysis of System Resources and Workload
- Tuning for Specific Query Types
- Basic Guidelines

Parallel Query Tuning Parameters

The extent to which parallel query processing is used is based on the tuning parameters listed in the following table. After these parameters are set, parallelism on demand is enabled whenever the specified conditions are met. Without any further tuning, the user benefits from parallel processing.

Parameter	Function
TUNE FILE_GROUP	Reduces seek contention on disk devices.
TUNE GROUP	Sets number of parallel tasks per file group (disk group).
TUNE TOTALQUERYPROCS	Sets maximum number of tasks available for parallel query processing at one time by all database server processes.
TUNE QUERYPROCS *	Sets maximum number of tasks available at one time for parallel query processing by a single database server process.
TUNE ROWS_PER_SCAN_TASK *	Limits number of parallel tasks applied to a query that performs a relation scan (that is, does not use an index).
TUNE ROWS_PER_JOIN_TASK *	Limits number of parallel tasks applied to the index-probing portion of a query.
TUNE ROWS_PER_FETCH_TASK *	Limits number of parallel tasks applied to the row-data-fetching phase of a query.
TUNE FORCE_FETCH_TASKS *	Sets the number of parallel tasks for fetching rows in queries that use a STAR index.
TUNE FORCE_JOIN_TASKS *	Sets the number of parallel tasks for joining tables in queries that use a STAR index.
TUNE FORCE_SCAN_TASKS *	Sets the number of parallel tasks for relation scans of tables.

* You can also set these parameters with a SET command anywhere that SQL statements can be entered.

(1 of 2)

Parameter	Function
TUNE FORCE_HASHJOIN_TASKS *	Sets the number of parallel tasks for hybrid hash joins.
TUNE FORCE_AGGREGATION_TASKS *	Sets the number of parallel tasks for partitioned parallel aggregation.
TUNE PARTIONED_PARALLEL_AGGREGATION *	Enables or disables parallelism for aggregation operations.

* You can also set these parameters with a SET command anywhere that SQL statements can be entered.

(2 of 2)

Enabling Parallel Query Processing

To enable parallel processing, you must set the QUERYPROCS and TOTALQUERYPROCS parameters in the configuration file (**rbw.config**) to a value greater than zero.

You can control the extent of parallelism for query processing in either of two ways:

- Specify the minimum number of rows you want a task to process with the ROWS_PER_TASK parameters, which avoid the overhead of parallel processing for trivial queries. These parameters are intended to handle the general purpose day-to-day processing.
- Specify the number of parallel tasks you want to use to process a query, regardless of the number of rows per task, with the FORCE_TASKS parameters. These parameters, which allow you more control over CPU resources, are designed for use on specific queries on which you want to specify the number of parallel tasks to be used in order to get the query processed quickly, regardless of the resources used. These parameters override the ROWS_PER_TASK parameters.

All parameters can be entered as TUNE parameters in the **rbw.config** file so that they affect all database server sessions. The order of these parameters in the **rbw.config** file is not significant. Alternatively, you can enter them as SQL SET statements, in which case they affect the current session only.

The following sections discuss the parallel tuning parameters in detail.

Limiting I/O Contention with the FILE_GROUP Parameter

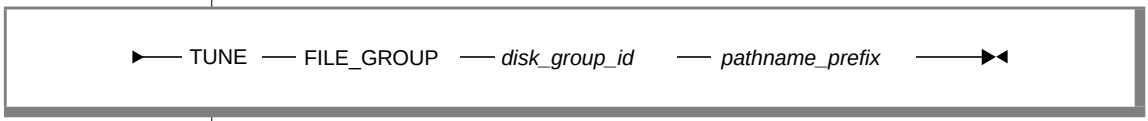
The FILE_GROUP parameter definitions specify to the database server what PSUs are on the same disk in order to reduce seek contention on individual disk devices. These groups of PSUs are referred to as *disk groups*. This parameter limits the amount of parallelism. In general, at most one task is allocated per disk group for each operation (such as a scan) unless the TUNE GROUP parameter is used to specify that more than one task can be used for a specific group.

Because of the SuperScan technology used for disk I/O, tasks from multiple database servers performing relation scans on tables can access the same data with a single read operation, which can reduce seek contention across database server processes.

These entries are read at database server startup, so changes are in effect for all sessions started after a change to the **rbw.config** file.

Syntax

To specify disk groups, enter a line for each disk group in the **rbw.config** file using the following syntax.



disk_group_id	Identifier used to group files into the same disk group. This variable must be an integer in the range of 1 to 32,767.
pathname_prefix	Any string of characters delimited by white space or end-of-line.

Usage Notes

To determine what group a file belongs to, the filename (as specified in CREATE SEGMENT statements) is converted to an absolute filename (corresponding on UNIX to either a link or an actual file). Then the longest pathname prefix that is a left substring of the filename is located. The *disk_group_id* value associated with this pathname prefix is the ID of the group to which the file belongs.

If no matching pathname prefix exists for a particular filename, the file is considered to be in its own private group with a limit of one task applied to that group.

UNIX

You can change this behavior by including an entry in the **rbw.config** file with a pathname prefix of '/' and a unique disk group ID. This entry matches any file not matched by any other entry. ♦

Any number of pathname prefixes can be mapped to the same disk group. Each mapping requires a separate TUNE FILE_GROUP statement.

If multiple TUNE FILE_GROUP statements contain identical pathname prefixes, the last such entry is used, and all others are discarded. With the exception of duplicate elimination, the order of the entries has no impact.

To allow more than one task per disk group, increase this limit with the TUNE GROUP parameter.

Examples

UNIX

A table has its data segmented as follows:

```
seg1            seg2            seg3
----------------------------------------
/disk1/psu11   /disk3/psu21   /disk4/psu31
/disk2/psu12   /disk4/psu22   /disk5/psu32
```

To reduce seek contention by placing PSUs on the same physical disk in the same file group, enter the following lines in the **rbw.config** file:

```
TUNE FILE_GROUP 1 /disk1
TUNE FILE_GROUP 2 /disk2
TUNE FILE_GROUP 3 /disk3
TUNE FILE_GROUP 4 /disk4
TUNE FILE_GROUP 5 /disk5
```

FILE_GROUP 4 contains both **psu22** and **psu31**, which are on the same disk (**/disk4**). Putting these PSUs in the same disk group prevents a query process from assigning two processes to work simultaneously on the two PSUs. ♦

WIN NT/2000

A table has its data segmented as follows:

```
seg1       seg2        seg3
----------------------------
f:\psu11   h:\psu21   i:\psu31
g:\psu12   i:\psu22   j:\psu32
```

To reduce seek contention by placing PSUs on the same physical disk in the same file group, enter the following lines in the **rbw.config** file:

```
TUNE FILE_GROUP 1 f:\
TUNE FILE_GROUP 2 g:\
TUNE FILE_GROUP 3 h:\
TUNE FILE_GROUP 4 i:\
TUNE FILE_GROUP 5 j:\
```

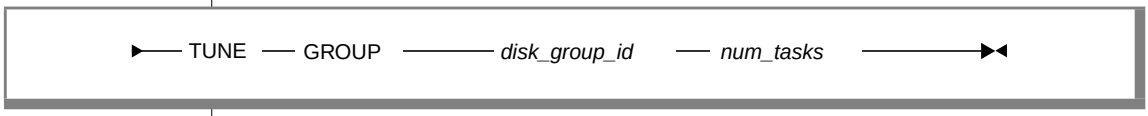
FILE_GROUP 4 contains both **psu22** and **psu31**, which are on the same disk (**i**). Putting these PSUs in the same disk group prevents a query process from assigning two tasks to work simultaneously on the two PSUs. ♦

Allowing Parallelism Within Disk Groups with the GROUP Parameter

The GROUP parameter provides a way to allow some parallel processing to occur within a disk group. In those cases where PSUs are striped across multiple disks, such as disk arrays or multiple disks grouped together as logical volumes, you can use this parameter to allow parallel processing for a specific disk group.

Syntax

To specify parallelism for a disk group, enter a line for each disk group in the **rbw.config** file using the following syntax.



disk_group_id Identifier used to define a disk group with the FILE GROUP parameter. This variable must be an integer value.

num_tasks Maximum number of outstanding tasks against the specified disk group. If you do not specify a GROUP parameter for a disk group, the default is one task at a time for that disk group.

Usage Notes

Even though the FILE_GROUP parameter is designed specifically to limit I/O contention within a specific disk group, multiple disks that are striped to appear as one logical volume to the operating system can support more I/O activity. You can increase parallel I/O activity to disk groups of this type (RAID disks or striped logical volumes) with the GROUP parameter.

In cases where queries are CPU intensive rather than I/O bound and there is excess CPU capacity, you can use the GROUP parameter to allow additional parallelism to take advantage of all the CPU capacity.

Enter a separate GROUP parameter for each disk group you want to modify.

Example

Suppose in the example on page 11-7 that **disk1** and **disk2** on UNIX or F and G on Windows are actually logical volumes that are striped across five physical disks each. To adjust the parallelism per disk group to accommodate CPU-intensive queries that are heavily concentrated in **seg1**, enter the following lines in the **rbw.config** file:

```
TUNE GROUP 1 5
TUNE GROUP 2 5
```

Any query that accesses **seg1** now might have as many as five I/O requests outstanding against **disk1** and **disk2** on UNIX or F and G on Windows.

Limiting Available Tasks

You can control the number of tasks available for parallel query processing and the allocation of those tasks in a multiuser environment. The parameter TOTALQUERYPROCS specifies the maximum number of tasks available for parallel queries at one time on all the database servers controlled by a single daemon, providing a mechanism to control the system load imposed by parallel queries. The QUERYPROCS parameter specifies the maximum number of concurrent parallel tasks to be used in processing a single query, providing a mechanism to control the resources allocated to a single database server (user).

The algorithm that allocates tasks to queries employs a *graceful decrease* mechanism to ration remaining tasks when the demand is high. After 50 percent of the total tasks available for processing queries have been allocated, subsequent queries are allocated fewer tasks per query.

Example

Assume TOTALQUERYPROCS is `1000`, QUERYPROCS is `100`, and 5 queries have each been allocated 100 tasks so that 500 tasks out of the 1000 total tasks have been allocated. In this case, a graceful decrease sets in, and subsequent queries each receive fewer than 100 tasks apiece (even if they request 100). Once the ratio of allocated tasks to total tasks drops below 50 percent, queries again receive the requested number of tasks, up to the limit imposed by the QUERYPROCS value.

TOTALQUERYPROCS

To specify a limit on the total number of tasks available for parallel queries to all database servers controlled by a single warehouse daemon, enter a line in the **rbw.config** file using the following syntax.



max_parallel_tasks A nonnegative integer in the range of 0 to 32,767. (A value of `0` or `1` effectively disables parallel query execution.)

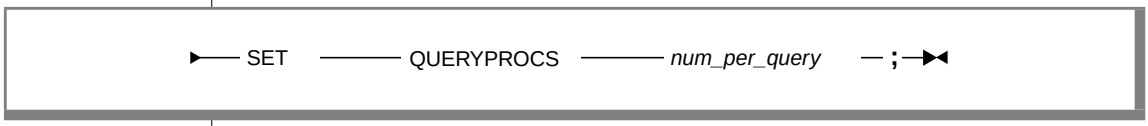
The count specified by *max_parallel_tasks* does not include tasks allocated to the base servers (the number limited by the MAX_SERVERS parameter). Changing this parameter might also require operating-system parameters to be changed, as described in the *Installation and Configuration Guide*. Individual operating systems have limits or operating-system parameters that limit the allowable range of TOTALQUERYPROCS.

QUERYPROCS

To specify a limit on the total number of parallel tasks available for a single query for all sessions, enter a line in the **rbw.config** file using the following syntax.



To specify a limit on the total number of parallel tasks available for a single query for specific sessions, enter a SET command using the following syntax.



num_per_query A nonnegative integer in the range of 0 to 32,767. Specifying a value of `0` effectively disables parallel query processing. This number is an upper limit. Other factors such as the ROWS_PER_TASK and FORCE_TASKS parameters and the TOTALQUERYPROCS parameter can also limit the number of tasks available for a single query. The distribution of data might also result in some tasks completing before others, which might reduce the amount of parallelism to less than expected.

Usage Notes for TOTALQUERYPROCS and QUERYPROCS

If either the TOTALQUERYPROCS parameter or the QUERYPROCS parameter is `0` or not present, no parallel processing is done.

If multiple TOTALQUERYPROCS or QUERYPROCS statements are present, the last such entry is used, and all others are discarded. With the exception of duplicate elimination, the order of the entries has no impact.

The TOTALQUERYPROCS entry in the **rbw.config** file is read at daemon startup, so changes are not effective until the **rbwapid** daemon is restarted.

The QUERYPROCS entry in the **rbw.config** file is read at database server startup, so changes are not effective until a new database server is started. If the SET command attempts to set QUERYPROCS to a value greater than the value in the **rbw.config** file, the value in the **rbw.config** file is used.

Setting Minimum Row Requirements with ROWS_PER_TASK Parameters

You can use the ROWS_PER_TASK parameters to provide limits to the database server on when to run processes in parallel. If a small number of rows is involved, the overhead of running the task in parallel might outweigh the benefit. You can set a ROWS_PER_TASK limit to instruct the database server not to start a parallel task unless the ROWS_PER_TASK value is greater than *x*. The following three parameters are used for different types of queries and at different points in query processing:

- The ROWS_PER_SCAN_TASK parameter sets a minimum on the number of rows that the database server expects to read in a relation scan before the scan is performed in parallel. This number does not affect queries that use an index, only those that perform a relation scan of the table.
- The ROWS_PER_FETCH_TASK parameter sets a minimum on the number of rows the database server estimates will be returned during the fetch portion of a STARjoin before performing the tasks in parallel.
- The ROWS_PER_JOIN_TASK parameter sets a minimum on the number of index entries the database server estimates will be returned during the join processing (index-probing) portion of a STARjoin before performing the tasks in parallel. The database server enables parallelism for both fetch and join tasks based on the number of join tasks it calculates, based on the equation in "Enabling Parallelism for a STARjoin" on page 11-19.

These parameters can be set for all sessions with entries in the **rbw.config** file. Changes are effective for database server sessions started after the change is made. If multiple values for a given parameter are specified in the configuration file, the last entry of each type is used, and all others are discarded. The order of the three ROWS_PER_TASK parameters has no impact. These parameters can also be set for a specific session with a SET command.

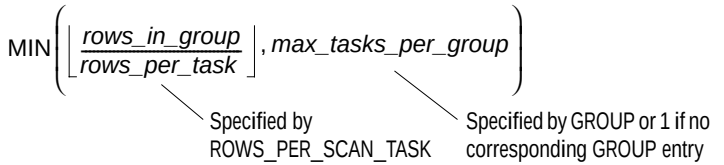
The following sections describe how to select values for these parameters. In general, the default values supplied with Red Brick Decision Server inhibit parallelism. If you are not satisfied with the resource consumption or query response time and feel that more or less parallelism would improve performance, adjust the values accordingly.

ROWS_PER_SCAN_TASK

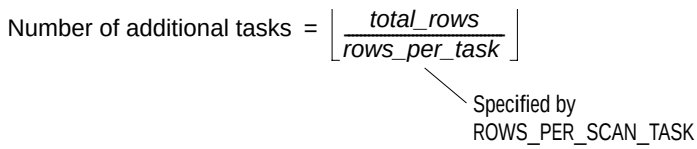
The ROWS_PER_SCAN_TASK parameter sets a limit on the number of parallel tasks initiated for a relation scan by specifying a minimum number of rows each scan task must return before it will run in parallel. This limit affects queries that use no index but scan an entire table.

The database server uses this parameter value as follows to determine the maximum number of tasks to use for a query of this type:

1. Each disk group with at least the number of rows specified by ROWS_PER_SCAN_TASK is assigned tasks based on the following formula:



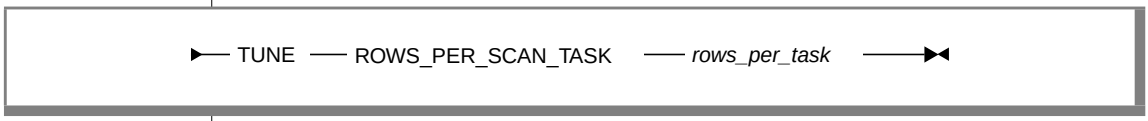
2. The number of rows in each disk group containing fewer than the specified number of rows are added together. This total row number is then divided by the number of rows specified by ROWS_PER_SCAN_TASK to determine how many additional tasks to allocate.



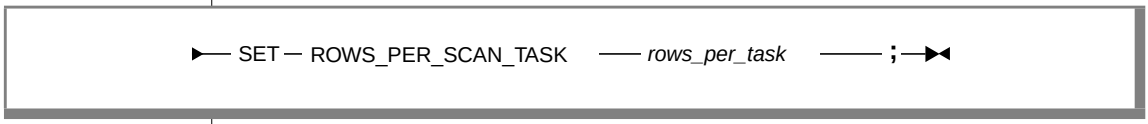
The number of rows (in a disk group) refers to the number of rows for which space has been allocated in a PSU, and this number might exceed the number of rows visible to a query. For example, space might be allocated for rows that have since been deleted.

Syntax

To specify the value used for *rows_per_task* in the preceding equation for all sessions, enter a line in the **rbw.config** file using the following syntax.



To specify the value used for *rows_per_task* in the preceding equation for specific sessions, enter a SET command using the following syntax.



rows_per_task An integer in the range of 1 to 2^{31}. A higher value provides less parallelism in returning rows from the queried table, and a lower value more parallelism. (Informix recommends this number be at least 5000.)

Example

Assume a table has space for 18,000,000 rows allocated across three segments, and each segment contains two PSUs. Each PSU is in its own disk group. The first PSU in each segment has been allocated up to its maximum size, which is sufficient to hold 4,500,000 rows, and the second PSU in each segment has sufficient space allocated to hold 1,500,000 rows. The following figure illustrates this table.

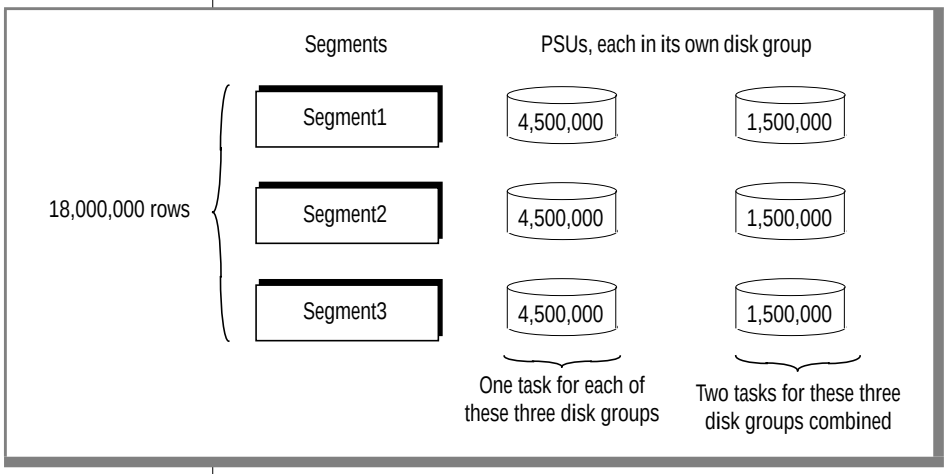


Figure 11-1 Disk Groups

Assuming a maximum of one task per disk group and a value of `2,000,000` for ROWS_PER_SCAN_TASK , the maximum number of tasks that could be allocated to a relation scan of the table is computed as follows:

1. Three disk groups, one for each of the large PSUs in each segment, are each large enough to hold over 2,000,000 rows. These disk groups contribute three tasks to the maximum task count even though each PSU is more than twice as large as the value for the parameter ROWS_PER_SCAN_TASK.
2. Three disk groups, one for each of the small PSUs in each segment, have allocated space insufficient to hold 2,000,000 rows. Thus the total number of rows that could be held in the allocated space in all these groups is added up to yield 4,500,000 rows. This number is divided by 2,000,000 and rounded down to yield two tasks.
3. The resulting maximum task count is 3 + 2 = **5**. So a relation scan of this table would be processed by at most 5 parallel tasks.

The number of tasks actually used is also bounded by the values set for the TOTALQUERYPROCS and QUERYPROCS parameters.

ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK

The database server uses the values of the ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK parameters to estimate the number of parallel tasks to use for queries that use a STAR index.

Queries vary in the amount of work done during the index-probing (join) phase and the row-data-processing (fetch) phase. These parameters allow you to set different limits for each phase. For example, if your queries tend to require a lot of processing after each row is fetched (GROUP BY, SUM, MIN, and so on), assign fewer rows per task for the fetch phase than for the join phase so that more tasks are used for the fetch phase.

These parameters allow you to control parallel processing for queries that use a STAR index, based on the following guidelines:

- The ROWS_PER_JOIN_TASK parameter enables parallelism for the STARjoin. If the number of join tasks is less than 1, parallelism is not enabled for both the fetch and join phases. For more details, refer to "Enabling Parallelism for a STARjoin" on page 11-19.
- The more tightly constrained a query is on the columns that participate in the STAR index, the smaller the number of parallel tasks needed to probe the index efficiently during the join phase. The ROWS_PER_JOIN_TASK parameter defines what *tight* is and how many tasks to use during the join phase.
- The more rows to be returned and the more processing of row data to be done, the larger the number of parallel tasks that can be used effectively during the fetch phase. The ROWS_PER_FETCH_TASK parameter determines how many tasks to use during the fetch phase.

The database server uses several equations with the values specified in these parameters to determine the degree of parallelism for the fetch and join phases:

- Estimated rows
- Number of tasks
- Parallelism enablement for STARjoin
- Row counts for dimension tables

The following sections describe these equations.

Estimated Rows

To estimate the number of rows to be processed by the join fetch and join phases for queries that use a STAR index, the database server uses the following equation:

$$\text{Estimated rows} = \frac{\langle \text{count(*) from fact\_table} \rangle \text{ X } \langle \Pi \text{rows selected from dimension tables} \rangle}{\Pi \langle \text{row counts for dimension tables} \rangle}$$

The *row counts for dimension tables* is either referenced row counts or total row counts. Referenced row counts are only calculated when the dimension table:

- Is the leading dimension in a STAR index on the fact table
- Is not a common dimension in a query that requires a multifact join

If either of these criteria is not met, the server calculates the selectivity by using the row count for the whole dimension table.

The numerator in this equation is applied after the constraints on the dimension tables are processed and the number of rows in each dimension table that satisfy the constraints is known.

Number of Tasks

The database server uses the estimated number of rows and the parameter ROWS_PER_TASK to calculate how many tasks to use during each phase of query processing. The following equations show this process:

$$\text{Number of fetch tasks} = \left\lfloor \frac{\text{estimated rows}}{\text{ROWS\_PER\_FETCH\_TASK}} \right\rfloor$$

$$\text{Number of join tasks} = \left\lfloor \frac{\text{estimated rows}}{\text{ROWS\_PER\_JOIN\_TASK}} \right\rfloor$$

Enabling Parallelism for a STARjoin

The database server executes a STARjoin in parallel only if it calculates one or more join tasks. In other words, the database server enables parallelism for a STARjoin when the following criterion is true:

```
Number of join tasks = estimated rows / ROWS_PER_JOIN_TASK >= 1
```



Important: *If the calculated number of join tasks is less than 1, the database server sets both the number of join tasks and the number of fetch tasks to 0, regardless of the calculated number of fetch tasks.*

Parallel fetches are enabled only if the following conditions are true:

- The calculated number of join tasks is 1 or more.

  If the calculated number of join tasks is less than 1, the calculated number of fetch tasks is ignored.
- The estimated number of rows is greater than the value of the ROWS_PER_FETCH_TASK parameter.

For an illustration of these criteria, see "Example 2: Enabling Parallelism for a STARjoin" on page 11-21.

Number of Rows in Dimension Tables

As the equation in "Estimated Rows" on page 11-17 shows, the database server counts the number of rows in each dimension table to estimate the number of rows to be processed by fetch and join tasks. The database server uses the ratio of the number of constrained rows in dimension tables to the total number of rows in dimension tables to derive the estimated rows.

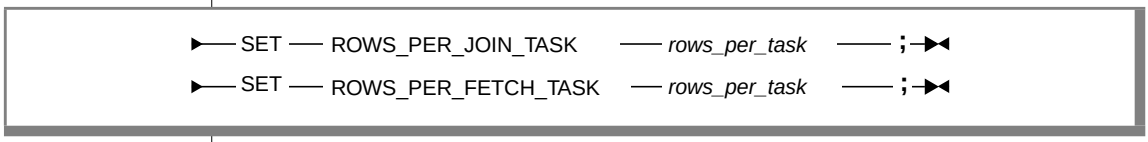
However, when a dimension table is the leading key of a STAR index, the estimation is based not on the total number of rows in the dimension but on the referenced row count, as described in Chapter 10, "Tuning a Warehouse for Performance." For optimal query performance, do not pre-load dimension tables unless such a STAR index exists. In general, the number of rows in a dimension should be close to the actual number of corresponding foreign key values in the referencing table. A large number of rows in a dimension table without corresponding foreign keys might disable parallel joins or result in the choice of a sub-optimal query plan.

Syntax

To specify the minimum number of rows per task used in the preceding equation for each phase for all sessions, enter lines in the **rbw.config** file using the following syntax.



To specify the minimum number of rows per task used in the preceding equation for each phase for specific sessions, enter a SET command using the following syntax.



rows_per_task Integers in the range of 1 to 2^{31}. A higher value provides less parallelism, and a lower value provides more parallelism. The database server does not run a query in parallel if the number of join tasks given by the equation in "Enabling Parallelism for a STARjoin" on page 11-19 is less than 1.



Tip: *As a general rule, the values of the parameters ROWS_PER_JOIN_TASK and ROWS_PER_FETCH _TASK should each be at least 5000 to justify the use of parallel tasks.*

Example 1: Estimating Rows

This example illustrates how the database server uses the equations in "Estimated Rows" on page 11-17 and "Number of Tasks" on page 11-18. A referencing (fact) table **Fact** has 3,000,000 rows. Three referenced (dimension) tables are used in its STAR index: **Product** with 2,000 rows, **Market** with 50 rows, and **Period** with 156 rows. The ROWS_PER_JOIN_TASK parameter is specified as 90,000 rows per task, and the ROWS_PER_FETCH_TASK parameter as 50,000.

After the constraints are processed, all the rows from the **Product** table, 10 rows from the **Market** table, and 52 rows from the **Period** table satisfy the constraints. Assume that the **Period** table is the leading dimension in a STAR index, so the referenced row count is calculated for that dimension, rather than the total row count for the table. In this case, suppose that the referenced row count is 78.

$$\text{Estimated rows} = \frac{(3,000,000) \times (2,000 \times 10 \times 52)}{(2,000 \times 50 \times 78)} = (400,000)$$

Based on ROWS_PER_JOIN_TASK, which is specified as 90,000 rows per task, apply the equation for the number of tasks used to process the STAR index as follows:

```
Number of join tasks = floor(400,000 / 90,000) = 4 tasks
```

Based on ROWS_PER_FETCH_TASK, which is specified as 50,000 rows per task, the number of tasks used to fetch the data and perform any result pre-aggregation is calculated as follows:

```
Number of fetch tasks = floor(400,000 / 50,000) = 8 tasks
```

These figures indicate that 12 (4 + 8) tasks could be used to process the query based on the values specified for the parameters ROWS_PER_JOIN_TASK and ROWS_PER_FETCH_TASK.

Example 2: Enabling Parallelism for a STARjoin

This example illustrates how the database server uses the equation in "Estimated Rows" on page 11-17 and the criteria in "Enabling Parallelism for a STARjoin" on page 11-19 to enable parallelism for a query that uses a STAR index.

Suppose a referencing fact table has 400,000,000 rows and two dimension tables involved in the STARjoin have 100 rows and 4000 rows, respectively. The number of rows that satisfy the constraints are 4 rows from one dimension table and 10 rows from the second dimension table. The dimension with 4000 rows is the leading dimension of a STAR index, hence the referenced row count of 2000 used in the equation rather than a total row count of 4000.

If ROWS_PER_JOIN_TASK is 600,000, the criteria for enabling parallel tasks for the STARjoin is not true, as the following equations show:

```
Estimated rows = (400,000,000) * ( (4 * 10) / (2000 * 100) )
               = 80,000

Number of join tasks = floor(Estimated rows / ROWS_PER_JOIN_TASK)
                     = 80,000 / 600,000
                     = 0 tasks

Criteria to enable parallelism:
      Number of join tasks >= 1 tasks
                       0 < 1
```

In this example, if ROWS_PER_FETCH_TASK is 20,000, the number of fetch tasks is 4, as the following equation shows. However, because the number of join tasks is less than 1, parallelism is not enabled for the fetch phase, even though the number of fetch tasks is 4.

```
Number of fetch tasks = floor(Estimated rows/ROWS_PER_FETCH_TASK)
                      = 80,000 / 20,000
                      = 4 tasks
```

On the other hand, if the referenced row count for the second dimension table is 100 instead of 2000, the database server enables parallel tasks, as the following equations show:

```
Estimated rows = (400,000,000) * ( (4 * 10) / (100 * 100) )
               = 1,600,000

Number of join tasks = floor(Estimated rows / ROWS_PER_JOIN_TASK)
                     = 1,600,000 / 600,000
                     = 2 tasks
```

Forcing the Number of Parallel Tasks with the FORCE_TASKS Parameters

The FORCE_TASKS parameters allow you to explicitly specify the number of parallel tasks that are used to process a query. The ROWS_PER_TASK parameters require that you determine a minimum number of rows needed to justify starting a parallel task, an implicit limit. Except for the parameter FORCE_HASHJOIN_TASKS, these parameters are analogous to the ROWS_PER_TASK parameters in their target queries:

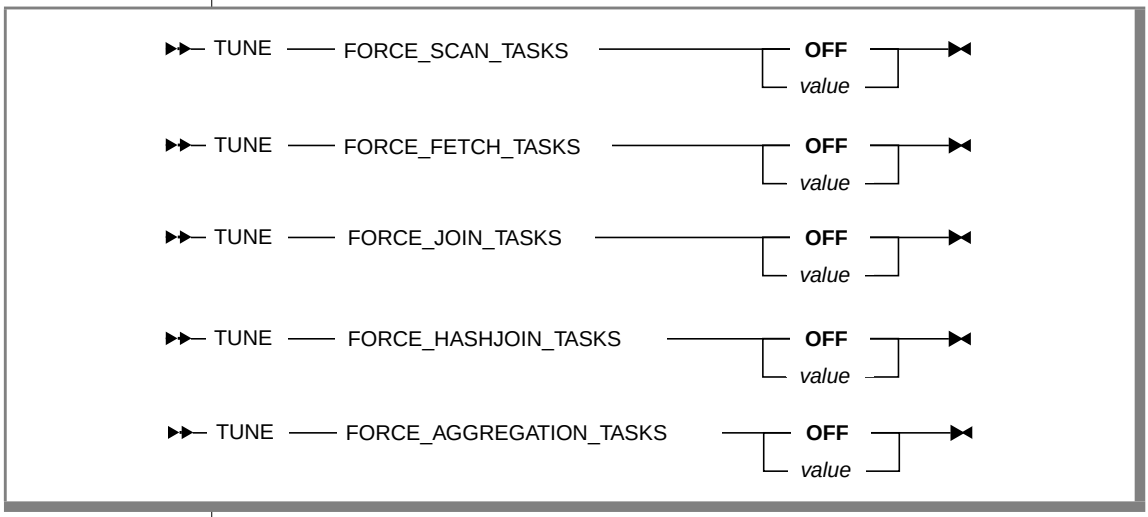
- FORCE_SCAN_TASKS specifies a maximum number of tasks that can be used for relation-scan operations. This parameter does not affect queries that use an index, only those that perform a relation scan.
- FORCE_FETCH_TASKS specifies the maximum number of parallel tasks that can be used for the fetch portion of a query using a STAR index.
- FORCE_JOIN_TASKS specifies the maximum number of parallel tasks that can be used for the join portion of a query using a STAR index.
- FORCE_HASHJOIN_TASKS specifies the maximum number of parallel tasks that can be used for each hybrid hash join in a query.
- FORCE_AGGREGATION_TASKS specifies the maximum number of parallel tasks that can be used for each group in a partitioned parallel query.

Use these parameters only in cases where you want to override the general allocation of parallel tasks. For example, you are running a query to build an aggregate table, no one else is using the system, and you want the query to complete as quickly as possible even if it greatly increases resource consumption.

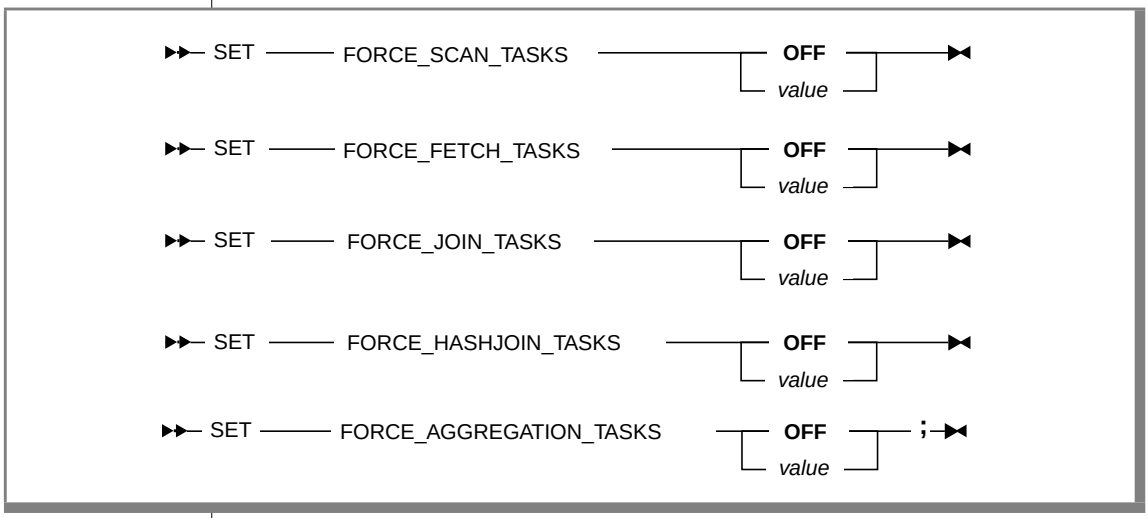
The FORCE_AGGREGATION_TASKS parameter has an effect only if partitioned parallelism is enabled with the PARTITIONED PARALLEL AGGREGATION parameter. For more information, refer to "Enabling Partitioned Parallelism for Aggregation" on page 11-31.

Syntax

To specify the number of tasks for all sessions, enter a line in the **rbw.config** file using the following syntax.



To specify the number of tasks for a single session, enter a SET command using the following syntax.



OFF

Explicit control of parallelism by the specified task limit is not enabled. The default value is OFF.

value

An integer value that explicitly limits the maximum number of tasks. However, this value does not guarantee the specified number. For the SCAN, FETCH, JOIN, and HASHJOIN parameters, the actual number of tasks used has an upper limit of the *lowest* of these three values:

- The specified FORCE_TASKS value
- The number of PSUs over which the table is distributed
- The number of tasks that can be allocated from the QUERYPROCS / TOTALQUERYPROCS pool

For the FORCE_AGGREGATION_TASKS parameter, the actual number of tasks used will have an upper limit of the *lower* of these two values:

- The specified FORCE_AGGREGATION_TASKS value.
- The number of tasks that can be allocated from the QUERYPROCS / TOTALQUERYPROCS pool.

No argument

If OFF or *value* is not specified, the SET command returns the current setting for that parameter. For example:

```
set force_scan_tasks;
** INFORMATION ** (1433) FORCE_SCAN_TASKS is
   currently set to 6.
```

FORCE_SCAN_TASKS

The value set for FORCE_SCAN_TASKS controls the number of parallel tasks for relation scans of tables.

However, the FORCE_SCAN_TASKS value does not guarantee that a certain number of parallel tasks is used. The actual number of tasks used is the lowest of these three values:

- The FORCE_SCAN_TASKS value
- The number of PSUs over which the table is distributed
- The number of tasks that can be allocated from the QUERYPROCS/TOTALQUERYPROCS pool

  After 50 percent of the TOTALQUERYPROCS pool has been allocated, subsequent queries are allocated fewer tasks per query.

Example

Assume the following settings.

Parameter	Value
FORCE_SCAN_TASKS	16
PSUs in table	18
QUERYPROCS	18
TOTALQUERYPROCS	24

Whether the FORCE_SCAN_TASKS value is used in this case depends on the number of tasks available from the TOTALQUERYPROCS pool. If only 6 tasks are already allocated, 18 tasks are available, so the FORCE_SCAN_TASKS value of 16 is used.

Usage Notes

The following points also apply to task allocation for relation scans:

- If FORCE_SCAN_TASKS is set, the ROWS_PER_SCAN_TASK value is ignored.
- If FORCE_SCAN_TASKS is set to a value that is greater than the number of disk groups, some disk groups are simply allocated more than one task. When FORCE_SCAN_TASKS is set, the number of disk groups does not influence the behavior of parallel processing.

FORCE_FETCH_TASKS and FORCE_JOIN_TASKS

The values set for FORCE_FETCH_TASKS and FORCE_JOIN_TASKS control the number of parallel tasks for fetching rows and joining tables in queries that use a STAR index. If either of these values is greater than or equal to 1, it overrides the corresponding value set for ROWS_PER_FETCH_TASK or ROWS_PER_JOIN_TASK.

However, the FORCE_FETCH_TASKS and FORCE_JOIN_TASKS values do not guarantee that a certain number of parallel tasks is used. The actual number of tasks used to fetch rows is the lowest of these three values:

- The FORCE_FETCH_TASKS value
- The number of PSUs over which the table is distributed
- The number of tasks available from the pool QUERYPROCS / TOTALQUERYPROCS

The actual number of tasks used to join tables is usually the lowest of these two values:

- The FORCE_JOIN_TASKS value
- The number of tasks available from the pool QUERYPROCS / TOTALQUERYPROCS

In rare cases, the FORCE_JOIN_TASKS value might be greater than the number of STAR index rows that match the constraints in the query. Therefore, it is not possible to logically divide and process the query by the specified number of tasks. Instead, the number of matching rows is used to set the limit on parallel join tasks.

Examples

FORCE_FETCH_TASKS

Assume the following settings.

Parameter	Value
FORCE_SCAN_TASKS	16
PSUs in table	18
QUERYPROCS	18
TOTALQUERYPROCS	24

In this case, whether the FORCE_FETCH_TASKS value is used depends on the number of tasks available from the TOTALQUERYPROCS pool. If only 6 tasks are already allocated, 18 tasks are available, and the FORCE_FETCH_TASKS value of 16 is used.

FORCE_JOIN_TASKS

Assume the following settings.

Parameter	Value
FORCE_SCAN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

If nine or more tasks are available from the TOTALQUERYPROCS pool, the FORCE_JOIN_TASKS value is used.

Usage Notes

The following points also apply to task allocation for fetching rows and joining tables:

- You do not have to force both fetch and join tasks. For example, you can force join tasks but allow fetch tasks to be computed dynamically.
- If FORCE_FETCH_TASKS is set, the ROWS_PER_FETCH_TASK value is not used. Similarly, if FORCE_JOIN_TASKS is set, the ROWS_PER_JOIN_TASK value is not used.
- Although the number of PSUs over which the table is distributed affects the allocation of parallel fetch tasks, the number of disk groups does not.
- The number of PSUs used to partition the STAR index does not affect the allocation of parallel join tasks.
- For joins of multifact tables, one fact table is selected to control the partitioning. If 10 PSUs are used to distribute the chosen fact table, 10 tasks are available for fetch-task partitioning.
- If fewer than the requested number of tasks are available from the QUERYPROCS/TOTALQUERYPROCS pool and both FORCE options are set, the system tries to preserve the ratio of FORCE_JOIN_TASKS to FORCE_FETCH_TASKS values.

FORCE_HASHJOIN_TASKS

The value set for FORCE_HASHJOIN_TASKS controls the number of parallel tasks for hybrid hash joins.

However, the FORCE_HASHJOIN_TASKS value does not guarantee that a certain number of parallel tasks is used. The actual number of tasks used is the *lowest* of the following values:

- The FORCE_HASHJOIN_TASKS value
- The number of tasks that can be allocated from the QUERYPROCS/TOTALQUERYPROCS pool

Example

Assume the following settings.

Parameter	Value
FORCE_SCAN_TASKS	8
QUERYPROCS	12
TOTALQUERYPROCS	30

If 10 or more tasks are available from the TOTALQUERYPROCS pool, the FORCE_HASHJOIN_TASKS value is used.

Usage Notes

The following points also apply to task allocation for parallel hybrid hash joins:

- The PARALLEL_HASHJOIN option must be set to ON, either with a SET PARALLEL_HASHJOIN ON command or a TUNE PARALLEL_HASHJOIN ON parameter, in order to gain any parallelism from hybrid hash joins.
- You must have at least 2 more than the value you specify in FORCE_HASHJOIN_TASKS available from the QUERYPROCS/TOTALQUERYPROCS pool in order to achieve that level of parallelism. For example, in order to get 8 parallel hash join tasks, you must specify a FORCE_HASHJOIN_TASKS value of 8 and have at least 10 tasks available from the pool QUERYPROCS/TOTALQUERYPROCS.

Enabling Partitioned Parallelism for Aggregation

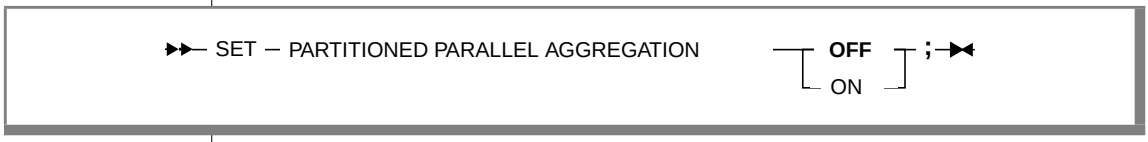
Queries that involve aggregation (SUM, MIN, MAX, COUNT) of groups specified in the GROUP BY clause can benefit from parallelism partitioned on the grouping columns, especially if there is a large number of groups (for example, hundreds of thousands or millions).

Syntax

To enable or disable partitioned parallel aggregation, enter a line in the **rbw.config** file using the following syntax.



To enable or disable partitioned parallel aggregation, enter a SET command using the following syntax.



OFF Disables partitioned aggregation parallelism. The default value is OFF.

Usage Notes

Once partitioned parallelism is enabled (set PARTITIONED PARALLEL AGGREGATION to ON), the value of the FORCE_AGGREGATION_TASKS parameter is enforced. If partitioned parallelism is disabled (PARTITIONED PARALLEL AGGREGATION set to OFF), the FORCE_AGGREGATION_TASKS parameter is ignored.

When PARTITIONED PARALLEL AGGREGATION is set to OFF, a potential performance gain occurs with parallel aggregation, but where the parallelism is not partitioned by the grouping columns. For relatively small numbers of groups, this approach should provide better performance than the partitioned parallelism.

When you set partitioned parallel aggregation on, the number of tasks used on your computer potentially doubles: the number of tasks needed for the parallel aggregation plus the number of tasks needed for the rest of the query processing. Therefore, to ensure the same resources to the other parts of the parallel query, increase the values of the parameters QUERYPROCS and TOTALQUERYPROCS.

Partitioned parallelism is also effective when populating a table with an INSERT INTO...SELECT...GROUP BY operation, and it is particularly effective when there is a large number of groups. This approach might improve the performance of these types of operations when populating aggregate tables for use with Informix Vista. For details about Vista, refer to the *Informix Vista User's Guide*.

System Considerations for Parallel Tasks

The parameters described in "ROWS_PER_FETCH_TASK and ROWS_PER_JOIN_TASK" on page 11-16 determine upper limits on the number of parallel tasks applied for specific operation. However, other system considerations also limit these numbers.

The number of tasks applied to index-join tasks is also limited by the number of file groups in which the segments for the selected STAR index reside. In order for two tasks to be used to process the join tasks, the index must be spread over at least two file groups. If it is fewer than two file groups, only one task will be applied (unless you force parallelism with the parameter FORCE_JOIN_TASKS or specify more than one disk group task with the parameter GROUP). In general, if you examine the segments in which the STAR index resides, the limit on the amount of parallelism used in the join tasks is the number of file groups covered by those segments.

The number of tasks assigned to fetch tasks is also limited in the same manner by the number of groups in which the data resides.

Another limitation is the possibility that the system might be unable to allocate the number of tasks desired. In "Example 1: Estimating Rows" on page 11-20, the equations indicate that six tasks should be allocated to process the query. However, if the system is able to allocate only four tasks (if other users are using some of the tasks allocated for processing parallel queries), the system must allocate the available tasks between the join and fetch phases. The allocation of limited tasks to the fetch phases is based on the following equation:

$$\text{Number of fetch processes} = \text{MAX}\left(1, \left\lfloor \frac{\text{requested for fetch}}{\text{total requested}} \times \text{total available} \right\rfloor\right)$$

The remaining available tasks are then allocated for join processing.

In the example, with only four tasks available, the number of tasks allocated to fetch processing is two.

$$\text{Number of fetch processes} = \text{MAX}\left(1, \left\lfloor \frac{3}{5} ¥ 4 \right\rfloor\right) = 2f$$

The remaining two tasks are allocated to join processing.

The most important point is that parallelism is limited by the number of groups in which the index and data reside. In general, the system does not allocate more tasks than there are groups affected by the query.

Example

This example illustrates how the number of tasks allocated is affected by the distribution of the index and data across file groups. Assume a table is implemented as follows:

```
create segment idx1 … (two PSUs);
create segment idx2 … (two PSUs);
create segment data1 … (three PSUs);
create segment data2 … (three PSUs);

create table fact…
data in (data1, data2) segment by …
primary index in (idx1, idx2) segment by references of (prodkey)
   ranges (min:1000, 1000:max)
```

Assume that each PSU is in a disk group by itself.

The maximum number of four tasks can be allocated for join processing because the index covers four file groups. If the constraints cover only one segment, the maximum number of join tasks actually would be two because the segment contains two PSUs, each in a separate disk group.

The row data accesses are unpredictable. However, the calculated maximum number of fetch tasks that might be allocated to fetch processing is six because the data is distributed across six PSUs, each in a separate file group.

Analysis of System Resources and Workload

Parallelism takes advantage of available system resources by scheduling more work concurrently to speed up query processing. Parallelism also introduces additional factors that affect performance gains. Getting the best performance on parallel queries depends on the degree to which you can exploit concurrency and load balancing across a system, a task that requires careful planning, even before you load the database. The main questions concern system resource allocation and usage.

Disk Usage

To reduce disk contention and increase I/O activity, the I/O load should be distributed evenly across as many physical disks as there are parallel tasks. The object is to improve disk service time and I/O concurrency in order to reduce the time a query waits because its tasks are blocked on I/O.

Ideally, no more than one disk group, as defined by the FILE_GROUP parameter, should be assigned to one physical disk. Disk arrays or disks grouped together using a logical volume manager facility are exceptions. In these cases, multiple physical disks are always grouped together as one logical disk. In the following discussions, a disk group is treated as interchangeable with a physical disk. If there are multiple files (PSUs) per physical disk, these files should normally be organized in a single disk group using the FILE_GROUP parameter. Such an organization ensures that I/O can be scheduled evenly and reduces excessive head movement on the disk, especially important for parallel scans where it is desirable to have an orderly sequential access on the file and to take as much advantage as possible of any available read-ahead capability.

In some cases where disk arrays or disks are grouped together, such as striped disks or RAID systems, or where the query workload is CPU intensive rather than I/O bound and there is excess CPU capacity, better performance might result from allowing more than one task per disk group. To allow parallel tasks within a disk group for a query, use the TUNE GROUP parameter to specify the number of tasks.

For Data

If you are not sure how uniformly the data will be distributed across segments, or if you want to spread the data across more physical disks because you anticipate the queries will tend to cluster on a single disk or only a few disks, use the SEGMENT BY HASH option in creating tables. Hashing segments helps to ensure that data will be evenly distributed across all segments and thereby helps to balance processing across parallel tasks. However, before deciding to use the hash option, consider the space management issues associated with hashing. For example, hashed segments cannot be dropped individually or taken offline for loads.

For STAR Indexes

It is not always necessary or desirable to segment the STAR index. Consider whether the benefit merits the extra administration overhead. Parallelism for join tasks is derived based on PSUs and disk groups. As long as there are multiple PSUs and they are not grouped in a disk group, parallelism can be invoked even if the STAR index is in a single segment.

If the STAR index is to be loaded across multiple segments for parallel join processing, load the index on separate disks if possible. As a minimum, allow as many index PSUs or segments as the planned number of join tasks. If the CPUs are fast, you might need to allow more index and join tasks to keep the CPUs busy. If not enough disks are available, the next best option is to load the index segments on the same disks as the table segments, as long as the disks are not too busy.

Memory Usage

Memory requirements increase with parallel processing, even if the number of users remains the same. Because parallel tasks are spawned from the *parent* server, the child tasks inherit many of the characteristics of the parent, including memory requirements. So be aware of the additional memory demands in parallel processing, particularly important in a multiuser environment.

To determine memory requirements at your site, first determine how many users are actively executing queries at the same time (concurrently active users). The number of concurrently active users determines the memory demand and paging/swapping activities in the system at any one time. For example, if 100 users are connected, but 80 of them submit a query (10 minutes long) only once a day, 18 of them only twice a day (5 minutes long), and 2 of them all the time, you can estimate about 3 concurrently active users (assuming the 98 users submit their queries evenly throughout the day). Three users probably will not introduce heavy paging or swapping activities in the system. However, if there are 50 concurrently active users, examine carefully whether there is enough memory to support them. If there is not enough memory, thrashing occurs, resulting in heavy paging or swapping activities.

If the number of concurrently active users is high, consider either adding more memory or reducing the amount of parallelism. You can reduce parallelism either by reducing the number of query tasks per user (QUERYPROCS) or by ensuring that only large queries use parallel processing. You can prevent trivial or small queries from using parallel processing by choosing a large value (thousands to tens-of-thousands for STARjoin queries and tens to hundreds-of-thousands for scans) in the ROWS_PER JOIN, FETCH, and SCAN_TASK parameters.

When the system is up and running, monitor the paging and swapping activities. Acceptable values for these two activities vary depending on system size and speed. Rather than using generic values that might apply to your system, monitor the system when it is not performing optimally. On UNIX, monitor the waiting for I/O percentage (WIO) in the System Activity Report (SAR) or an equivalent performance monitoring tool, and also monitor the disk service time. On Windows, to monitor the percentage of time spent waiting for I/O and the disk service time, use Performance Monitor or another performance monitoring tool.

You can also evaluate disk usage based on busy percentage, disk request queue, and disk waiting time. If all these indicators are high, users are probably kept waiting while the system is busy paging or thrashing in memory. To decrease the wait time, you can add more paging and swapping devices, reduce parallelism, or add memory.

CPU Allocation

The degree of concurrency in parallel processing largely depends on how many CPUs are available. Although allocating all the CPUs in the system might yield the best results, this option is often not practical because other work in the system could be competing for CPU resources. If all the CPUs are used for parallel query processing, other users would experience a slow-down because CPU resources are less available to them. Therefore, each administrator must decide how the CPU resources are to be distributed. Of course, if all the CPUs are already saturated (that is, 100 percent busy), the result is no gain and perhaps even a slight degradation from parallel processing.

After you have determined the number of CPUs to use for parallel query processing, you can derive the number of parallel tasks (QUERYPROCS) as follows:

- For a query that is CPU intensive, set the QUERYPROCS parameter to the number of CPUs divided by the number of concurrently active users.
- For a query that is not CPU intensive but is I/O intensive, set the QUERYPROCS parameter to two or three times the value derived for CPU-intensive queries. Monitor the CPU-busy statistic to determine whether more parallel tasks are needed.

For example, assume you have a 12-CPU system and you want to allocate about 65 percent of CPU resources (8 CPUs) to parallel processing. If the system is 45 percent busy overall, it is reasonable to add more parallel tasks until the system reaches at least 65 percent busy. And if other tasks are executing in the system at the same time, it is reasonable to add even more parallel tasks, depending on the distribution of resource consumption. (If there are other tasks, parallel processing probably was not consuming 65 percent of the CPU resources.)

The FILE_GROUPS setup also affects concurrency. For I/O-intensive queries, specify as many disk groups (physical disks) as there are parallel tasks (QUERYPROCS). One factor that the database server uses to determine how many parallel tasks to create is the number of disk groups. If the number of disk groups is less than the QUERYPROCS value, the number of parallel tasks to create is reduced to the number of disk groups.

If there are more CPUS than disk groups and there is excessive CPU capacity, use the TUNE GROUP parameter to provide more parallelism than is normally allocated per disk group.

Tuning for Specific Query Types

This section describes how you can fine-tune parallel processing for specific query types. Before tuning for parallel processing, however, make sure that the queries themselves are fine-tuned with well-written SQL.

The FORCE_TASKS parameters override the corresponding ROWS_PER_TASK parameters and are not intended for use as general tuning parameters. The following discussion focuses on tuning with the ROWS_PER_TASK parameters.

For information on how Red Brick Decision Server processes queries, refer to "Understanding Query Processing" on page 10-44.

Parallel STARjoin Queries

The speedup expected from parallel processing of STARjoin queries (queries that use a STAR index) depends primarily on:

- Density of the query (number of rows selected from the referencing table)
- Number of parallel tasks
- Mix of parallel (join and fetch) tasks

- Number of disk groups, or file groups, as defined by the TUNE FILE_GROUPS entries in the **rbw.config** file
- Amount of parallelism per disk group, as defined by the TUNE GROUP entries in the **rbw.config** file

  Allowing parallelism within disk groups might increase disk contention and degrade performance.

Density

In general, the higher the density of a query, the better the potential for speedup.

Number of Parallel Tasks

The number of tasks can depend on the distribution of the data. If data for the query is clustered in the referencing table, it is possible that not all parallel tasks will get work. In this case, hashing the referencing table segments helps distribute the data more evenly across disks. For some cases where STARjoin queries are slow, alternate STAR indexes might provide more speedup than parallelism. Depending on which columns are constrained, consider alternate STAR indexes before resorting to parallel processing.

Mix of Parallel Tasks and File Groups

Choosing the right mix between join and fetch tasks is also important in speedup gains. For example, assume a query requires relatively more post processing than join processing but has only one fetch task allocated. The parallel speedup is much less than if more fetch tasks were allocated.

Use the following guidelines to finely tune the mix for specific queries:

- For queries dominated by post processing, allocate more fetch than join tasks.
- For queries dominated by join processing, allocate more join than fetch tasks.
- For a mix of join and post processing, start by specifying an equal number of join and fetch tasks. Then alter the ratio in both directions by 50 percent to determine which ratio is best. Usually, ratios of $n/1$ or $1/n$ are not the best choices, but there are exceptions.

Use the ROWS_PER_JOIN / FETCH_TASK parameters to control the ratio between join and fetch tasks. The values for these two parameters are inversely proportional to the ratio desired. For example, for six join and two fetch tasks, specify:

```
rows per join task = 2000
rows per fetch task = 6000
```

The actual allocation of tasks is more complicated:

1. The number of tasks is calculated by dividing the rows per join / fetch task into the estimated number of rows. If the join or fetch value is less than the estimated number of rows, no parallel tasks are created.
2. The number of tasks is evaluated against the number of file groups and group limits. The smaller number becomes the new number of tasks. For join tasks, the number of file groups for index segments is used. For fetch tasks, the number of file groups for the table is used.
3. The QUERYPROCS value is compared with the sum of the join and fetch tasks. The smaller number becomes the new number of tasks.

   If the QUERYPROCS value is smaller, the ratio between the join and fetch tasks (as specified by the ROWS_PER_JOIN / FETCH_TASK values) is preserved. However, if the number of file groups for either joins or fetches is less than the number of tasks allocated by the ratio, the ratio is not preserved. Maximum parallelism is offered by allocating join and fetch tasks up to the QUERYPROCS value.
4. If the remaining number of tasks specified by the QUERYPROCS value is less than 50 percent of TOTALQUERYPROCS, a graceful degradation process begins, assigning only a portion of the remaining QUERYPROCS number.

Considerations for Multiuser Environments

In a multiuser environment, it is often not worthwhile to offer parallelism for queries that take less than one minute or so. Parallel queries consume much more memory, and the user-perceivable speed improvement is relatively small. Consider the corresponding resource trade-off to decide which query types (small, medium, or large) merit parallel processing. To prevent small queries from using parallel processing, specify high values for the ROWS_PER_JOIN / FETCH_TASK parameters.

Additionally, use the TOTALQUERYPROCS parameter to limit the number of users of parallel processing. A reasonable value for the TOTALQUERYPROCS parameter is two to three times the QUERYPROCS value. If there is enough memory, you might find values of five times or higher acceptable. High values for the ROWS_PER_JOIN / FETCH_TASK parameters ensure that only large queries use parallel processing.

Parallel Table Scans

The speedup provided by parallel processing of table-scan operations is determined by the number of parallel scan tasks, the number of file (disk) groups available, and the TUNE GROUP parameter value.

Depending on the speed of the CPUs, several parallel scan tasks might be required to keep the number of CPUs busy. The recommended procedure is to assign no more than one disk group to one physical disk and to allow parallel scan processing to assign no more than one task per file (disk) group, thus facilitating sequential contiguous disk access whenever no more than one user is accessing the disk at a time. Otherwise, disk service times per block might increase by orders of magnitude. However, you can use the TUNE GROUP parameter to allow more than one task per group in cases where the query is CPU bound on a multiprocessor system with additional CPU capacity.

As described for parallel STARjoin queries, the recommendations for limiting parallelism for small queries (those that take one minute or less) in multiuser environments apply to ROWS_PER_SCAN_TASK as well.

SuperScan Technology

SuperScan technology is used whenever multiple users are scanning the same table at the same time or overlap some of the time. The perceived improvement varies greatly from time to time, depending on system activities.

SuperScan technology takes advantage of the file buffer cache for the operating system to reduce physical I/O activity. If the scanning tasks find most of the I/O blocks in the file buffer cache, there will be a large decrease in I/O activity. If few blocks are found in the file buffer cache, there will be a smaller decrease. Because the file buffer cache is used by everyone in the system, its state (that is, the hit rate) depends on the system activities. The interval between additional users beginning to scan the table also affects the decrease in I/O activity. If the timing and patterns of usage vary from day to day, improvement from SuperScan technology might vary.

About Reasonable Values

No one set of tuning values applies to all customers across all queries. As discussed in the previous sections, many considerations affect parallelism, and they differ for each customer, depending on the hardware platform, configuration, query mix, and number of users. The information presented here highlights general areas of concern and provides guidelines for tuning query performance for your environment.

The ROWS_PER_JOIN/FETCH/SCAN_TASK parameters have two main uses. First, they control parallelism and determine when it is invoked. Use these parameters to set the minimum estimated number of rows required before parallel processing is used. Secondly, the ROWS_PER_JOIN/FETCH_TASK parameters determine the ratio between join and fetch tasks for STARjoin queries.

Previous sections discuss factors relevant to the number of parallel tasks. The primary one is the number of CPUs. Then disk groups, memory, and users are considered, with TOTALQUERYPROCS limiting the number of users for parallel query processing.

Basic Guidelines

The following suggestions provide some basic guidelines for setting reasonable values for parallel processing. For detailed information, refer to the previous sections. Each system and workload are unique, so you must experiment to determine what works best at your site:

1. Set the ROWS_PER_JOIN / FETCH / SCAN_TASK values fairly high. For more information, refer to "Memory Usage" on page 11-36 and "Setting Minimum Row Requirements with ROWS_PER_TASK Parameters" on page 11-12.
2. Set ROWS_PER_JOIN_TASK and ROWS_PER_FETCH_TASK to the same value. For more information, refer to "Tuning for Specific Query Types" on page 11-38.
3. Set QUERYPROCS as discussed in "CPU Allocation" on page 11-37.
4. Set TOTALQUERYPROCS to at least two to three times the QUERYPROCS value, or even higher if memory is available. For more information, refer to "Limiting Available Tasks" on page 11-10.
5. If multiple PSUs for the same table are on the same disk device, define a disk group (file group) for those PSUs to limit disk contention. For more information, refer to "Disk Usage" on page 11-34.
6. In most cases, you do not need to set the TUNE GROUP parameter. Generally, only one task per disk group is best. For more information, refer to "Allowing Parallelism Within Disk Groups with the GROUP Parameter" on page 11-8.

These basic settings provide users with a reasonable environment for parallelism on large queries, but they do not process every query optimally. To fine-tune parallel processing for more optimal processing of specific queries, refer to the suggestions in "Tuning for Specific Query Types" on page 11-38.

Appendix

A

Example: Building a Database

This appendix illustrates how to build a database using **Aroma**, the sample database included with Red Brick Decision Server. It includes the following sections:

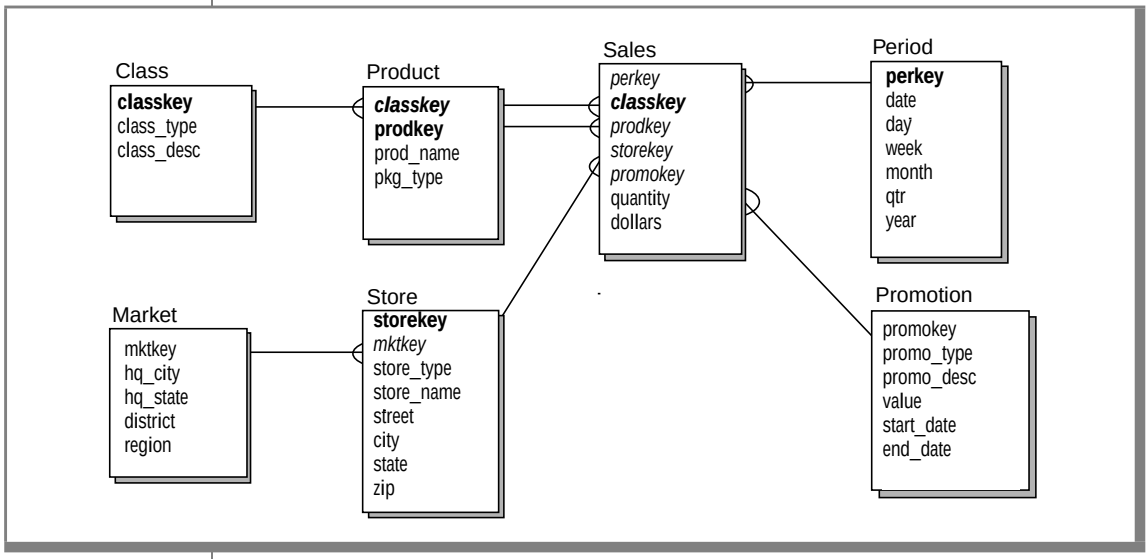
- Building the Aroma Database
- Logging In as redbrick
- Making the Database Directory
- Creating the Database
- Changing the Default Password
- Creating the User Tables
- Writing the LOAD DATA Statements
- Loading the Data
- Verifying the Database
- Summary

Building the Aroma Database

This example assumes the database schema has been defined and focuses on implementing that schema. The example uses the **Aroma** sample database included with Red Brick Decision Server. Because **Aroma** is a relatively small database, default segments are used for the dimension (referenced) tables, and named segments are used for the referencing (fact) table **Sales**. However, before you build production databases, perform a careful analysis of the space requirements and the anticipated database modifications and load patterns to determine whether to use named segments.

The basic **Aroma** database contains seven tables: **Sales**, **Class**, **Product**, **Market**, **Store**, **Promotion**, and **Period**, as the following figure illustrates.

Figure A-1
Schema of Aroma Database



For more information on the **Aroma** database, refer to the *SQL Self-Study Guide*.

The redbrick Directory and Aroma Input Files

The **Aroma** database input files are included on the media containing Red Brick Decision Server. Install them on your system in the directory ***redbrick_dir*/sample_input** on UNIX or ***redbrick_dir*\SAMPLE_INPUT** on Windows, where ***redbrick_dir*** is the Red Brick Decision Server directory on your system.

The procedures that follow assume you are going to copy the **Aroma** input files from the ***redbrick_dir*/sample_input** or ***redbrick_dir*\SAMPLE_INPUT** directory to a directory named **aroma_inputs** on UNIX or **AROMA_INPUTS** on Windows, which you create in your own directory. However, you can also use the files directly from those directories. Or you can use any editor to write your own files from the examples shown in this appendix.

Steps for Building a Database

To build the **Aroma** database, follow these steps:

1. Log in as the **redbrick** user.
2. Make a directory for the new database. Use the operating-system command.
3. Create the database. Use **rb_creator** on UNIX or **dbcreate** on Windows.
4. Change the default password for the database. Use the RISQL Entry Tool, GRANT statement.
5. Create the user tables for the database. Use the RISQL Entry Tool, CREATE TABLE statements.
6. Write the LOAD DATA statements and control files for the Table Management Utility (TMU). Use any editor.
7. Load the data into the database. Use the **rb_tmu** or **rb_ptmu** script.
8. Verify that the database was built and loaded successfully.

The following sections describe each step.

Logging In as redbrick

The **redbrick** user is the administrative account for Red Brick Decision Server. You must be logged in as **redbrick** for the administrative tasks involved in creating a database.

UNIX

The ***redbrick_dir*/bin** directory must also be in the **redbrick** user's path.

To log in as redbrick

1. Log in as the **redbrick** user. To ensure that permissions are set correctly, you must be the **redbrick** user to build the database. If you do not know the password, see your database administrator.
2. Verify that the ***redbrick_dir*/bin** directory is in your path.

   ```
   $ env
   ```

   In the list of environment variables displayed, check the entry for PATH. Its definition should include ***redbrick_dir*/bin**. ♦

WIN NT/2000

Alternately, the **redbrick** user or the Windows administrator can grant any user execute permission for **dbcreate**. The ***redbrick_dir*\BIN** directory must also be in the **redbrick** user's path.

To log in as redbrick

1. Log in as the **redbrick** user. If you do not know the password, see your database administrator.
2. Verify that the ***redbrick_dir*\BIN** directory is in your path.

   ```
   c:\> set
   ```

   In the list of environment variables displayed, check the entry for PATH. Its definition should include ***redbrick_dir*\BIN**. ♦

Making the Database Directory

Determine where you want to build the database. Then create two directories, one for the database and one for the input files, and copy the **Aroma** input files. The **Aroma** scripts, inputs, and loaded database require about 16 megabytes of storage.

UNIX

To create the database directory

1. As the **redbrick** user, change to the directory you have selected (**my_directory** in this example).

   ```
   $ cd my_directory
   ```

   This will be the parent directory for your new database.

2. From the parent directory (**my_directory** in this example), create a database directory named **aroma_db**.

   ```
   $ mkdir aroma_db
   ```

3. Verify that permissions on the **aroma_db** directory are set correctly.

   ```
   $ ls -dl aroma_db
   ```

   The permissions, as a minimum, should be as follows to indicate that the **redbrick** user has read, write, and execute permissions:

   ```
   drwx------ aroma_db
   ```

   Permissions for group and other depend on your environment and are not important for this exercise.

4. Create a directory named **aroma_inputs** for the **Aroma** input files.

   ```
   $ mkdir aroma_inputs
   ```

   By keeping the input files in a separate directory, you can more easily determine what files are created during each step.

5. Copy the **Aroma** files to the **aroma_inputs** directory.

   ```
   $ cp redbrick_dir/sample_input/aroma* aroma_inputs
   ```

♦

WIN NT/2000

To create the database directory

1. Change to the directory you have selected (**my_directory** in this example).

   ```
   c:\> cd my_directory
   ```

   This will be the parent directory for your database.

2. From the parent directory (**my_directory** in this example), create a database directory named **aroma_db**.

   ```
   c:\> mkdir aroma_db
   ```

3. Create a directory named **aroma_inputs** for the **Aroma** input files.

   ```
   c:\> mkdir aroma_inputs
   ```

   By keeping the input files in a separate directory, you can more easily determine what files are created during each step.

4. Copy the **Aroma** files to the **aroma_inputs** directory.

   ```
   c:\> copy redbrick_dir\sample_input\aroma* aroma_inputs
   ```

♦

The sample database directories and files look like this.

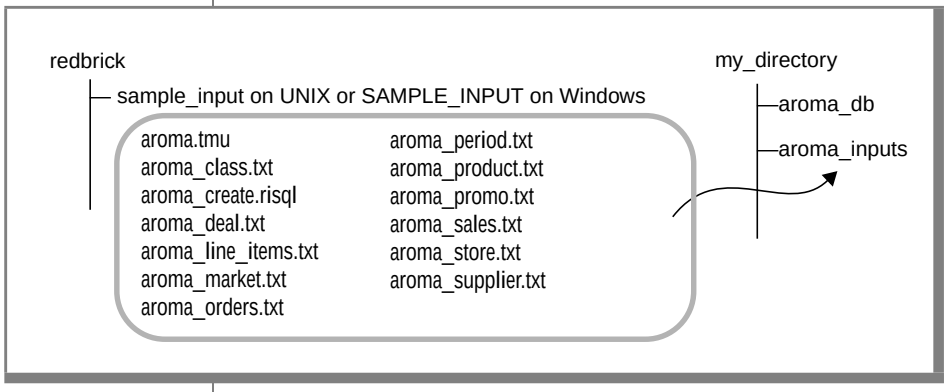


Figure A-2
Sample Database

Creating the Database

Create the new database using the **rb_creator** create script on UNIX or the **dbcreate** script on Windows. This step creates the system tables. You must be the **redbrick** user to execute **rb_creator** or **dbcreate**.

To create the database

1. Verify that you are in the directory containing the database.

Operating System	Command
UNIX	$ cd my_directory
Windows	c:>\cd my_directory

2. Run the create script.

UNIX

```
$ rb_creator aroma_db
```

Enter a logical database name for the new database in the **rbw.config** file, using any text editor. In this example, the new database is named **Newaroma**.

```
DB NEWAROMA /my_directory/aroma_db
```

♦

WIN NT/2000

```
c:>\ dbcreate -create -d aroma_db -l NEWAROMA
```

This script creates a database with the logical name **Newaroma** and adds the following line to the **rbw.config** file:

```
DB NEWAROMA c:\my_directory\aroma_db
```

♦

To access **Newaroma,** use the **-d** option on the command line for the RISQL Entry Tool, RISQL Reporter, or the TMU. You can omit the **-d** option if the **RB_PATH** environment variable is set to the logical database name for the desired database.

3. Log out of the **redbrick** account.

The **rb_creator** or **dbcreate** script creates the database system files, **RB_DEFAULT_IDX**, **RB_DEFAULT_INDEXES**, **RB_DEFAULT_LOCKS**, **RB_DEFAULTS_SEGMENTS**, and **RB_DEFAULT_TABLES**, in the **aroma_db** directory.

Changing the Default Password

Change the default password for **system**, the administrative account automatically created for each new database, from **manager** to a secure password. In this example, it is changed to **cryptic**.

To change the default password

1. Verify that the ***redbrick_dir*/bin** directory is in your path on UNIX or that the ***redbrick_dir*\bin** directory is in your path on Windows.
2. Invoke the RISQL Entry Tool, providing the logical database name and the default user name and password.

Operating System	Command
UNIX	`$ risql -d NEWAROMA system manager`
Windows	`c:\> risql -d NEWAROMA system manager`

   When you invoke the RISQL Entry Tool, the **RB_CONFIG** environment variable must point to the configuration file **rbw.config**. For more information, refer to the *RISQL Entry Tool and RISQL Reporter User's Guide*.

3. Change the password for the system account from manager to **cryptic**.

```
RISQL> grant connect to system with cryptic;
RISQL>
```

Creating the User Tables

User tables are created by writing and executing CREATE TABLE statements, which are based on the tables and columns defined in the database schema. This example uses default segments, named segments, automatic indexes, and user-created indexes. It does not use views or synonyms.

CREATE TABLE Statements

The CREATE TABLE statements for the **Aroma** database are in the file **aroma_create.risql**. The statements for the **Market**, **Store**, **Class**, **Product**, **Promotion**, **Period**, and **Sales** tables are as follows:

```
create table market (
   mktkey integer not null,
   hq_city char(20) not null,
   hq_state char(20) not null,
   district char(20) not null,
   region char(20) not null,
   constraint mkt_pkc primary key (mktkey));

create table store (
   storekey integer not null,
   mktkey integer not null,
   store_type char(10) not null,
   store_name char(30) not null,
   street char(30) not null,
   city char(20) not null,
   state char(5) not null,
   zip char (10) not null,
   constraint store_pkc primary key (storekey),
   constraint store_fkc foreign key (mktkey)
      references market (mktkey))
   maxrows per segment 2500;

create table class (
   classkey integer not null,
   class_type char (12) not null,
   class_desc char(60) not null,
   primary key (classkey));

create table product (
   classkey integer not null,
   prodkey integer not null,
   prod_name char(30) not null,
   pkg_type char(20) not null,
   constraint prod_pkc primary key (classkey, prodkey),
   constraint prod_fkc foreign key (classkey)
      references class (classkey))
   maxrows per segment 2500;

create table promotion (
   promokey integer not null,
   promo_type integer not null,
   promo_desc char(40) not null,
   value dec(7,2) not null,
   start_date date not null,
   end_date date not null,
   primary key (promokey))
   maxrows per segment 2500;
```

```
create table period (
   perkey integer not null,
   date date not null,
   day character(3) not null,
   week integer not null,
   month character(5) not null,
   qtr character(5) not null,
   year integer not null,
   primary key (perkey))
   maxrows per segment 2500;

create table sales (
   perkey integer not null,
   classkey integer not null,
   prodkey integer not null,
   storekey integer not null,
   promokey integer not null,
   quantity integer not null,
   dollars dec(7,2) not null,
   constraint sales_pkc primary key
      (perkey, classkey, prodkey, storekey, promokey),
   constraint sales_date_fkc foreign key (perkey)
      references period (perkey),
   constraint sales_product_fkc foreign key (classkey, prodkey)
      references product (classkey, prodkey),
   constraint sales_store_fkc foreign key (storekey)
      references store (storekey),
   constraint sales_promo_fkc foreign key (promokey)
      references promotion (promokey))
   data in (daily_data1, daily_data2)
      segment by values of (perkey)
      ranges (min:415, 415:max)
   maxsegments 2
   maxrows per segment 60000;
```

The following points apply to CREATE TABLE statements:

- Referenced tables are defined before any tables that reference them.
- Referenced table statements include MAXROWS PER SEGMENT values.
- The primary key and foreign key columns are declared NOT NULL.
- To facilitate creation of hierarchies, all columns for all tables are declared NOT NULL for the **Aroma** database.

- Each column in a table has a declared data type that corresponds to the data to be stored. **Newaroma** uses the following data types:
  - CHARACTER, which contains the specified number of characters
  - INTEGER (4 bytes)
  - DECIMAL (7,2), with a precision of 7 and a scale of 2 (4 bytes)
  - DATE (3 bytes)
- The **Sales** table is created using named segments.
- The other tables are all created using default segments.

For more information about planning and creating databases, refer to Chapter 4, "Planning a Database Implementation," and Chapter 5, "Creating a Database."

For a detailed description of the CREATE TABLE syntax and more information about data types, refer to the *SQL Reference Guide*.

To create the user tables for Newaroma

1. To invoke the RISQL Entry Tool, enter the following command at the system prompt:

   ```
   risql -d NEWAROMA system
   ```

   Provide the logical database name and user name.

2. Supply your password in response to the prompt.

   ```
   (C) Copyright 1991 - 2000, Informix, Los Gatos,
   California, USA
   All rights reserved
   RISQL Entry Tool Version 6.1 (xxxx)
   Please type password:
   RISQL>
   ```



Important: *To maintain password security, do not enter your password on the command line. Instead, enter it when you are prompted to do so.*

3. From the RISQL Entry Tool, read and execute the statements contained in the **aroma_create.risql** file.

Operating System	Command
UNIX	`RISQL> run /my_directory/aroma_inputs/aroma_create.risql;`
Windows	`RISQL> run \my_directory\aroma_inputs\aroma_create.risql;`

You can also enter the statements interactively from the command line, but running them from a file that you can edit is much easier and subject to fewer errors.

The **aroma_create.risql** script creates tables, automatically creates primary key B-TREE indexes on the tables, creates segments for the **Sales** table, drops the primary key B-TREE index on the **Sales** table, creates a STAR index, and creates a TARGET index. The remaining steps in this section show how to verify that they were built and how to obtain additional information about the tables and indexes that you might need later for various administrative tasks.

4. To verify that the tables were created, query the system table **RBW_TABLES**. Enter a SQL statement similar to the following:

```
RISQL> select * from rbw_tables where id > 0;
```

If the tables were created, the response is similar to the following one (although the **DATETIME**, **SEGMENT_BY**, **PARTIAL** and **COMMENT** columns will also be displayed, and the lines will wrap).

```
RISQL> select * from rbw_tables where id > 0;
Or
RISQL> select substr(name,1,13)as NAME, type, substr(creator,1,8) as CREATOR, id, maxseg
ments, maxrows_per_seg, maxsize_rows, intact from rbw_tables where id > 0;

NAME         TYPE    CREATOR  ID    MAXSEGMENTS MAXROWS_PER MAXSIZE_ROW INTA
ORDERS       TABLE   SYSTEM     11           1        2000    25165728 Y
SALES        TABLE   SYSTEM     13           2       60000      144102 Y
PERIOD       TABLE   SYSTEM      8           1        2500    79429329 Y
SUPPLIER     TABLE   SYSTEM      7           1        NULL    17825724 Y
STORE        TABLE   SYSTEM      2           1        2500    18612153 Y
PROMOTION    TABLE   SYSTEM      5           1        2500    36175734 Y
PRODUCT      TABLE   SYSTEM      4           1        2500    36175734 Y
LINE_ITEMS   TABLE   SYSTEM     12           1        2000    65011464 Y
CLASS        TABLE   SYSTEM      3           1        NULL    27787158 Y
MARKET       TABLE   SYSTEM      1           1        NULL    25165728 Y
```

Note the following about the response:

- The system tables were excluded by specifying `id > 0`.
- MAXROWS PER SEGMENT and MAXSEGMENTS values are specified for the tables in which they are defined.

5. To verify that the indexes were created automatically, enter the following statement:

```
RISQL> select * from rbw_indexes ;
```

If the indexes were created, the response is similar to the following (although the **DATETIME**, **INTACT**, **PARTIAL**, and **COMMENT** columns will also be displayed, and the lines might wrap).

```
RISQL> select * from rbw_indexes;
Or
RISQL> select substr(name,1,20) as NAME, substr(tname,1,12) as TNAME, type,
substr(cname,1,12) as COLUMN_NAME, substr(creator,1,10) as CREATOR, fillfactor,
state from rbw_indexes;
NAME                 TNAME      TYPE    COLUMN_NAME CREATOR FILLFACTOR STATE
MARKET_PK_IDX        MARKET     BTREE   MKTKEY       SYSTEM   100       VALID
STORE_PK_IDX         STORE      BTREE   STOREKEY     SYSTEM   100       VALID
CLASS_PK_IDX         CLASS      BTREE   CLASSKEY     SYSTEM   100       VALID
PRODUCT_PK_IDX       PRODUCT    BTREE   CLASSKEY     SYSTEM   100       VALID
PROMOTION_PK_IDX     PROMOTION  BTREE   PROMOKEY     SYSTEM   100       VALID
DEAL_PK_IDX          DEAL       BTREE   DEALKEY      SYSTEM   100       VALID
SUPPLIER_PK_IDX      SUPPLIER   BTREE   SUPKEY       SYSTEM   100       VALID
PERIOD_PK_IDX        PERIOD     BTREE   PERKEY       SYSTEM   100       VALID
ORDERS_PK_IDX        ORDERS     BTREE   ORDER_NO     SYSTEM   100       VALID
LINE_ITEMS_PK_IDX    LINE_ITEMS BTREE   ORDER_NO     SYSTEM   100       VALID
SALES_STAR_IDX       SALES      STAR    PERKEY       SYSTEM   100       VALID
STORE_TGT_IDX        STORE      TARGETS STORE_TYPE   SYSTEM   100       VALID
STORE_FK_IDX         STORE      BTREE   MKTKEY       SYSTEM   100       VALID
PRODUCT_FK_IDX       PRODUCT    BTREE   CLASSKEY     SYSTEM   100       VALID
ORDERS_FK1_IDX       ORDERS     BTREE   PERKEY       SYSTEM   100       VALID
ORDERS_FK2_IDX       ORDERS     BTREE   SUPKEY       SYSTEM   100       VALID
ORDERS_FK3_IDX       ORDERS     BTREE   DEALKEY      SYSTEM   100       VALID
LINE_ITEMS_STAR_IDX LINE_ITEMS STAR    ORDER_NO     SYSTEM   100       VALID
```

The following points apply to the indexes:

- The primary key B-TREE indexes are created automatically when the tables are created.
- The other indexes are created with CREATE INDEX statements, which are included in the **aroma_create.risql** file.
- Except for the **Sales** table, each table has a B-TREE index on its primary key. (The **aroma_create.risql** file dropped the primary key index on the **Sales** table.) The **Sales** table has a STAR index on its foreign keys. The **Store** table has a TARGET index on the **Store_Type** column.

6. Check the database directory to see what files were created.

Operating System	Command
UNIX	`RISQL>!ls /my_directory/aroma_db ;`
Windows	`RISQL>!dir /w \my_directory\aroma_db ;`

The exclamation mark (!) is an escape to the system shell on UNIX or to the MS-DOS shell on Windows. You should see a list similar to the following:

```
RB_DEFAULT_IDX       dfltseg18_psu1    dfltseg31_psu1
RB_DEFAULT_INDEXES   dfltseg19_psu1    dfltseg32_psu1
RB_DEFAULT_LOCKS     dfltseg1_psu1     dfltseg3_psu1
RB_DEFAULT_SEGMENTS  dfltseg20_psu1    dfltseg4_psu1
RB_DEFAULT_TABLES    dfltseg21_psu1    dfltseg5_psu1
dfltseg10_psu1       dfltseg22_psu1    dfltseg6_psu1
dfltseg11_psu1       dfltseg23_psu1    dfltseg7_psu1
dfltseg12_psu1       dfltseg24_psu1    dfltseg8_psu1
dfltseg13_psu1       dfltseg25_psu1    dfltseg9_psu1
dfltseg14_psu1       dfltseg26_psu1    sales_psu1
dfltseg15_psu1       dfltseg27_psu1    sales_psu2
dfltseg16_psu1       dfltseg28_psu1    sales_psu3
dfltseg17_psu1       dfltseg2_psu1     sales_psu4
```

If you do not use default segments for the tables and indexes, the PSUs are in the locations that the CREATE SEGMENT statements specify (or in default directories that the **rbw.config** file specifies). The PSUs for the **Sales** table are also in this directory. The CREATE SEGMENT statements (in the **aroma_create.risql** file) specify that the segments be created in the directory from which the script is run.

7. You can determine the table or index to which each segment belongs.

```
RISQL> select name, tname, iname, NPSUS from rbw_segments;
```

The response is similar to the following one (although the lines might wrap).

```
RISQL> select name, tname, iname, npsus from rbw_segments;
Or
RISQL> select substr(name, 1,20) as name, substr(tname,1,17) as tname,
substr(iname, 1,22) as iname, NPSUS from rbw_segments;
NAME                 TNAME            INAME                NPSUS
RBW_SYSTEM           NULL             NULL                    5
DEFAULT_SEGMENT_1    MARKET           NULL                    1
DEFAULT_SEGMENT_2    MARKET           MARKET_PK_IDX           1
DEFAULT_SEGMENT_3    STORE            NULL                    1
DEFAULT_SEGMENT_4    STORE            STORE_PK_IDX            1
DEFAULT_SEGMENT_5    CLASS            NULL                    1
DEFAULT_SEGMENT_6    CLASS            CLASS_PK_IDX            1
DEFAULT_SEGMENT_7    PRODUCT          NULL                    1
DEFAULT_SEGMENT_8    PRODUCT          PRODUCT_PK_IDX          1
DEFAULT_SEGMENT_9    PROMOTION        NULL                    1
DEFAULT_SEGMENT_10   PROMOTION        PROMOTION_PK_IDX        1
DEFAULT_SEGMENT_11   DEAL             NULL                    1
DEFAULT_SEGMENT_12   DEAL             DEAL_PK_IDX             1
DEFAULT_SEGMENT_13   SUPPLIER         NULL                    1
DEFAULT_SEGMENT_14   SUPPLIER         SUPPLIER_PK_IDX         1
DEFAULT_SEGMENT_15   PERIOD           NULL                    1
DEFAULT_SEGMENT_16   PERIOD           PERIOD_PK_IDX           1
DEFAULT_SEGMENT_17   ORDERS           NULL                    1
DEFAULT_SEGMENT_18   ORDERS           ORDERS_PK_IDX           1
DEFAULT_SEGMENT_19   LINE_ITEMS       NULL                    1
DEFAULT_SEGMENT_20   LINE_ITEMS       LINE_ITEMS_PK_IDX       1
DAILY_DATA1          SALES            NULL                    2
DAILY_DATA2          SALES            NULL                    2
DEFAULT_SEGMENT_23   SALES            SALES_STAR_IDX          1
DEFAULT_SEGMENT_24   STORE            STORE_TGT_IDX           1
DEFAULT_SEGMENT_25   STORE            STORE_FK_IDX            1
DEFAULT_SEGMENT_26   PRODUCT          PRODUCT_FK_IDX          1
DEFAULT_SEGMENT_27   ORDERS           ORDERS_FK1_IDX          1
DEFAULT_SEGMENT_28   ORDERS           ORDERS_FK2_IDX          1
DEFAULT_SEGMENT_29   ORDERS           ORDERS_FK3_IDX          1
DEFAULT_SEGMENT_30   LINE_ITEMS       LINE_ITEMS_STAR_IDX     1
RISQL>
```

The following points apply to segments:

- Default segments are used for all the tables except the **Sales** table. The system automatically assigns the numeric suffixes on these names. The CREATE SEGMENT and CREATE TABLE statements specify the segments for the **Sales** table.
- The **RBW_STORAGE** system table contains additional information about each PSU.

Writing the LOAD DATA Statements

The TMU uses LOAD DATA statements to map the input data from the input record fields to the corresponding table columns. Each table requires its own LOAD DATA statement, based on the input file and record formats and the table definitions. This example assumes all input data is in operating-system disk (not tape) files.

A single control file can contain multiple LOAD DATA statements, or each statement can be in a separate file. The LOAD DATA statements for each table in **Aroma** are in a single file named **aroma.tmu**.

This section displays both the data and the CREATE TABLE statements, followed by the LOAD DATA statements for the **Period**, **Product**, **Market**, and **Sales** tables in **Newaroma**. You can use the LOAD DATA statements provided, so you do not need to write them. Just look at them to see how they were derived from the input data and the CREATE TABLE statements.

The Period Table

Input data records for the **Period** table (in the file **aroma_period.txt**) look like this:

```
1*1998-01-01*SA*1*JAN*Q1_98*1998
2*1998-01-02*SU*2*JAN*Q1_98*1998
3*1998-01-03*MO*2*JAN*Q1_98*1998
4*1998-01-04*TU*2*JAN*Q1_98*1998
5*1998-01-05*WE*2*JAN*Q1_98*1998
6*1998-01-06*TH*2*JAN*Q1_98*1998
7*1998-01-07*FR*2*JAN*Q1_98*1998
8*1998-01-08*SA*2*JAN*Q1_98*1998
9*1998-01-09*SU*3*JAN*Q1_98*1998
```

```
10*1998-01-10*MO*3*JAN*Q1_98*1998
11*1998-01-11*TU*3*JAN*Q1_98*1998
12*1998-01-12*WE*3*JAN*Q1_98*1998
13*1998-01-13*TH*3*JAN*Q1_98*1998
14*1998-01-14*FR*3*JAN*Q1_98*1998
15*1998-01-15*SA*3*JAN*Q1_98*1998
16*1998-01-16*SU*4*JAN*Q1_98*1998
...
```

This is only a partial list of the input records.

This input data is stored in the **Period** table that the following CREATE TABLE statement defines:

```
create table period (
   perkey integer not null,
   date date not null,
   day character(3) not null,
   week integer not null,
   month character(5) not null,
   qtr character(5) not null,
   year integer not null,
   primary key (perkey))
   maxrows per segment 2500;
```

The LOAD DATA statement that reads each input data record and maps each field in the record to a column in the corresponding row in the **Period** table looks like this:

```
load data inputfile 'aroma_period.txt'
   replace
   format separated by '*'
   into table period (
      perkey integer external (4),
      date date 'YYYY-MM-DD',
      day char(3),
      week integer external (4),
      month char(5),
      qtr char(5),
      year integer external);
```

The following points apply to the LOAD DATA statement for the **Period** table:

- The records use a separated format with an asterisk (*) separator.
- The load operation is performed in REPLACE mode. Any existing data in the table is destroyed.
- The first field has the field type **integer external**, is four characters long, and is to be stored in the **Period** table column named **Perkey**.

- The second field has the field type **date** and has a format mask that specifies four digits for year and two digits for month and day. A dash (-) separates subfields.
- The third field has the field type **character**, is three characters long, and is to be stored in the **Period** table column named **Day**.

The Product Table

Input data records for the **Product** table (in the file **aroma_product.txt**) look like this:

```
1:00:Veracruzano :No pkg
1:01:Xalapa Lapa :No pkg
1:10:Colombiano  :No pkg
1:11:Expresso XO :No pkg
1:12:La Antigua  :No pkg
1:20:Lotta Latte :No pkg
1:21:Cafe Au Lait:No pkg
1:22:NA Lite :No pkg
1:30:Aroma Roma :No pkg
1:31:Demitasse Ms:No pkg
2:00:Darjeeling Number 1 :No pkg
2:01:Darjeeling Special :No pkg
2:10:Assam Grade A  :No pkg
2:11:Assam Gold Blend :No pkg
2:12:Earl Grey:No pkg
```

This is only a partial list of the input records.

This data is stored in the **Product** table that the following CREATE TABLE statement defines:

```
create table product (
   classkey integer not null,
   prodkey integer not null,
   prod_name char(30) not null,
   pkg_type char(20) not null,
   constraint prod_pkc primary key (classkey, prodkey),
   constraint prod_fkc foreign key (classkey)
      references class (classkey))
   maxrows per segment 2500;
```

The LOAD DATA statement that reads each input data record and maps each field in the record to a column in the corresponding row in the **Product** table looks like this.

```
load data
   inputfile 'aroma_product.txt'
   replace
   format separated by ':'
   discardfile 'product.discards'
   discards 1
   into table product (
      classkey integer external(2),
      prodkey integer external(2),
      prod_name char(30),
      pkg_type char(20)) ;
```

The following points apply to the LOAD DATA statement for the **Product** table:

- These data records use a separated format with a colon (:) separator.
- Discards are written to a file named **product.discards.** If a single record is discarded, the TMU terminates.
- Only character and external data types are present. Although a length parameter is specified for each field, it is ignored with separated-format records.

The Market Table

Input data records for the **Market** table (in the file **aroma_market.txt**) look like this.

```
01*Atlanta*GA*Atlanta*South
02*Miami*FL*Atlanta*South
03*New Orleans*LA*New Orleans*South
04*Houston*TX*New Orleans*South
05*New York*NY*New York*North
06*Philadelphia*PA*New York*North
07*Boston*MA*Boston*North
08*Hartford*CT*Boston*North
09*Chicago*IL*Chicago*Central
10*Detroit*MI*Chicago*Central
11*Minneapolis*MN*Minneapolis*Central
```

```
12*Milwaukee*WI*Minneapolis*Central
14*San Jose*CA*San Francisco*West
15*San Francisco*CA*San Francisco*West
16*Oakland*CA*San Francisco*West
17*Los Angeles*CA*Los Angeles*West
19*Phoenix*AZ*Los Angeles*West
```

This is only a partial list of the input records.

This data is stored in the **Market** table that the following CREATE TABLE statement defines:

```
create table market (
   mktkey integer not null,
   hq_city char(20) not null,
   hq_state char(20) not null,
   district char(20) not null,
   region char(20) not null,
   constraint mkt_pkc primary key (mktkey));
```

The LOAD DATA statement that reads each input data record and maps each field in the record into a column in the corresponding row in the **Market** table looks like this.

```
load data
   inputfile 'aroma_market.txt'
   replace
   format separated by '*'
   discardfile 'market.discards'
   discards 1
   into table market (
      mktkey integer external(2),
      hq_city char(20),
      hq_state char(2),
      district char(13),
      region char(7)) ;
```

In the LOAD DATA statement for the **Market** table, no RECORDLEN clause is specified, which allows the TMU to handle the variable-length records with newline-separated data.

The Sales Table

The input data records for the **Sales** table (in the file **aroma_sales.txt**) look like this.

```
00000000002 00000000002 00000000000 00000000001 00000000116 00000000008 000000034.00
00000000002 00000000004 00000000012 00000000001 00000000116 00000000009 000000060.75
00000000002 00000000001 00000000011 00000000001 00000000116 00000000040 000000270.00
00000000002 00000000002 00000000030 00000000001 00000000116 00000000016 000000036.00
00000000002 00000000005 00000000022 00000000001 00000000116 00000000011 000000030.25
00000000002 00000000001 00000000030 00000000001 00000000116 00000000030 000000187.50
00000000002 00000000001 00000000010 00000000001 00000000116 00000000025 000000143.75
00000000002 00000000004 00000000010 00000000002 00000000000 00000000012 000000087.00
00000000002 00000000004 00000000011 00000000002 00000000000 00000000014 000000115.50
00000000002 00000000002 00000000022 00000000002 00000000000 00000000018 000000058.50
00000000002 00000000004 00000000000 00000000002 00000000000 00000000017 000000136.00
00000000002 00000000005 00000000000 00000000002 00000000000 00000000013 000000074.75
00000000002 00000000004 00000000030 00000000002 00000000000 00000000014 000000101.50
00000000002 00000000002 00000000010 00000000002 00000000000 00000000018 000000063.00
00000000002 00000000001 00000000022 00000000003 00000000000 00000000011 000000099.00
00000000002 00000000006 00000000046 00000000003 00000000000 00000000006 000000036.00
00000000002 00000000005 00000000012 00000000003 00000000000 00000000010 000000040.00
```

This is only a partial list of the input records.

This input data is stored in the **Sales** table that the following CREATE TABLE statement defines:

```
create table sales (
   perkey integer not null,
   classkey integer not null,
   prodkey integer not null,
   storekey integer not null,
   promokey integer not null,
   quantity integer not null,
   dollars dec(7,2) not null,
   constraint sales_pkc primary key
      (perkey, classkey, prodkey, storekey, promokey),
   constraint sales_date_fkc foreign key (perkey)
      references period (perkey),
   constraint sales_product_fkc foreign key (classkey, prodkey)
      references product (classkey, prodkey),
   constraint sales_store_fkc foreign key (storekey)
      references store (storekey),
   constraint sales_promo_fkc foreign key (promokey)
      references promotion (promokey))
   data in (daily_data1, daily_data2)
      segment by values of (perkey)
      ranges (min:415, 415:max)
   maxsegments 2
   maxrows per segment 60000;
```

The LOAD DATA statement that reads each input data record and maps each field in the record to a column in the corresponding row in the **Sales** table looks like this.

```
load data inputfile 'aroma_sales.txt'
   recordlen 86
   insert
   into table sales (
     perkey position(2) integer external(11) nullif(1)='%',
     classkey position(14) integer external(11) nullif(13)='%',
     prodkey position(26) integer external(11) nullif(25)='%',
     storekey position(38) integer external(11) nullif(37)='%',
     promokey position(50) integer external(11) nullif(49)='%',
     quantity position(62) integer external(11) nullif(61)='%',
     dollars position(74) decimal external(12) nullif(73)='%');
```

The following points apply to the **Sales** table:

- These input data records are in fixed-format records. The position of each field is specified by a POSITION clause, and there are no separators.
- The RECORDLEN clause is specified, which fixes these records to the length 86. The total length of the individual fields (11+11+11+11+11+11+12) plus 1 character for each NULLIF (1+1+1+1+1+1+1) plus the newline character equals the record length (86).

Loading the Data

To load the **Newaroma** data, run the TMU with the control file that contains the LOAD DATA statements. In this database, all the LOAD DATA statements are combined into a single control file named **aroma.tmu**. The portions of that file for the **Period**, **Product**, **Market**, and **Sales** table look like this.

```
load data inputfile 'aroma_period.txt'
   replace
   format separated by '*'
   into table period (
      perkey integer external (4),
      date date 'YYYY-MM-DD',
      day char(3),
      week integer external (4),
      month char(5),
      qtr char(5),
      year integer external);

load data
   inputfile 'aroma_product.txt'
   replace
   format separated by ':'
   discardfile 'product.discards'
   discards 1
```

```
   into table product (
      classkey integer external(2),
      prodkey integer external(2),
      prod_name char(30),
      pkg_type char(20)) ;

load data
   inputfile 'aroma_market.txt'
   replace
   format separated by '*'
   discardfile 'market.discards'
   discards 1
   into table market (
      mktkey integer external(2),
      hq_city char(20),
      hq_state char(2),
      district char(13),
      region char(7)) ;

load data inputfile 'aroma_sales.txt'
 recordlen 86
 insert
 into table sales (
   perkey position(2) integer external(11) nullif(1)='%',
   classkey position(14) integer external(11) nullif(13)='%',
   prodkey position(26) integer external(11) nullif(25)='%',
   storekey position(38) integer external(11) nullif(37)='%',
   promokey position(50) integer external(11) nullif(49)='%',
   quantity position(62) integer external(11) nullif(61)='%',
   dollars position(74) decimal external(12) nullif(73)='%');
```

The following points apply to the LOAD DATA statements and the control file:

- The input file name must be relative to the directory from which you invoke the TMU, or it must be a full pathname.
- The referenced (dimension) tables (**Period**, **Product**, and **Market**) must be loaded before the referencing (fact) table (**Sales**).

To load the data into the aroma_db database

1. Log in as the **redbrick** user.
2. Verify that the directory *redbrick_dir*/**bin** on UNIX or *redbrick_dir***BIN** on Windows is in your path.
3. Change to the **aroma_inputs** directory, which contains the **aroma.tmu** file with the LOAD DATA statements for all the tables in the **Aroma** database.
4. Run the TMU.

Operating System	Command
UNIX	`$ rb_tmu -d /my_directory/aroma_db aroma.tmu system cryptic`
Windows	`c:\> rb_tmu -d \my_directory\aroma_db aroma.tmu system cryptic`

The TMU responds with messages similar to the following ones.

```
(C) Copyright 1991-2000 Informix Software,Inc.
All rights reserved.
Version 6.10.0(0)TST
** INFORMATION ** (366) Loading table MARKET.
** WARNING ** (8023) Any existing rows in tables that reference table MARKET may now be
invalid.
** INFORMATION ** (315) Finished file aroma_market.txt. 17 rows read from this file.
** INFORMATION ** (367) Rows: 17 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:00.05 cp time, 00:00:00.59 time, Logical IO count=90, Blk
Reads=5, Blk Writes=29
** INFORMATION ** (366) Loading table PRODUCT.
** WARNING ** (8023) Any existing rows in tables that reference table PRODUCT may now be
invalid.
** INFORMATION ** (315) Finished file aroma_product.txt. 59 rows read from this file.
** INFORMATION ** (367) Rows: 59 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:00.05 cp time, 00:00:00.81 time, Logical IO count=125, Blk
Reads=1, Blk Writes=48
** INFORMATION ** (366) Loading table PROMOTION.
** WARNING ** (8023) Any existing rows in tables that reference table PROMOTION may now be
invalid.
** INFORMATION ** (352) Row 102 of index PROMOTION_PK_IDX is out of sequence. Switching to
standard optimized index building. Loading continues...
** INFORMATION ** (315) Finished file aroma_promo.txt. 194 rows read from this file.
** INFORMATION ** (513) Starting merge phase of index building PROMOTION_PK_IDX.
** INFORMATION ** (367) Rows: 194 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:00.37 cp time, 00:00:00.90 time, Logical IO count=76, Blk
Reads=3, Blk Writes=36
** INFORMATION ** (366) Loading table PERIOD.
** WARNING ** (8023) Any existing rows in tables that reference table PERIOD may now be
invalid.
** INFORMATION ** (315) Finished file aroma_period.txt. 821 rows read from this file.
** INFORMATION ** (367) Rows: 821 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:00.05 cp time, 00:00:00.63 time, Logical IO count=87, Blk
Reads=4, Blk Writes=38
** INFORMATION ** (366) Loading table SALES.
** INFORMATION ** (352) Row 3 of index SALES_STAR_IDX is out of sequence. Switching to standard
optimized index building. Loading continues...
** INFORMATION ** (315) Finished file aroma_sales.txt. 69941 rows read from this file.
** INFORMATION ** (513) Starting merge phase of index building SALES_STAR_IDX.
** INFORMATION ** (367) Rows: 69941 inserted. 0 updated. 0 discarded. 0 skipped.
** INFORMATION ** (500) Time = 00:00:03.97 cp time, 00:00:08.63 time, Logical IO count=755, Blk
Reads=736, Blk Writes=699
```

The following points apply to the messages from the TMU:

- For each table, the TMU information reports the number of rows from the input file, which are categorized as inserted, updated, discarded, or skipped for the table. In this case, all rows were inserted.
- All tables were loaded, including the other tables in the **Aroma** database. The messages for the **Promotion** and **Sales** tables indicate that as their indexes were built, the TMU detected out-of-order data and switched to the mode called *standard optimized mode* to continue building the indexes. The input data is ordered with respect to the STAR index on the **Sales** table, based on the order of its leading foreign key reference. It is unordered with respect to the foreign key references of the other tables.

If you want other users to access this database, you must provide them database access with the **GRANT** statement. Also define the database with a logical name in the **rbw.config** file to simplify database selection.

Verifying the Database

To verify that the tables were built, enter a simple SELECT statement.

```
RISQL> select * from product where classkey = 1;
```

If the data is loaded correctly, the response is similar to this.

```
RISQL> select * from product where classkey = 1;
CLASSKEY    PRODKEY     PROD_NAME                      PKG_TYPE
          1           0 Veracruzano                    No pkg
          1           1 Xalapa Lapa                    No pkg
          1          10 Colombiano                     No pkg
          1          11 Expresso XO                    No pkg
          1          12 La Antigua                     No pkg
          1          20 Lotta Latte                    No pkg
          1          21 Cafe Au Lait                   No pkg
          1          22 NA Lite                        No pkg
          1          30 Aroma Roma                     No pkg
          1          31 Demitasse Ms                   No pkg
RISQL>
```

Summary

To summarize the steps in building a database:

1. Determine the tables needed for the database and the columns and data types for each table.
2. Log in as the **redbrick** user and create the database with **rb_creator** on UNIX or **dbcreate** on Windows. Add the logical database name to the **rbw.config** file.
3. Create the tables and indexes using SQL CREATE statements.
4. Review the format of the input data (if it already exists).
5. Write the LOAD DATA statements to map the external data representation to the internal format. You can then combine these LOAD DATA statements in a single TMU control file or run them individually.
6. As the **redbrick** user, load the data with **rb_tmu** or **rb_ptmu** and the control files.
7. Grant access to users.

Appendix

B

Configuration File

When Red Brick Decision Server is installed as directed in the *Installation and Configuration Guide*, the installation procedure creates a configuration file named **rbw.config** in the **redbrick** directory.

This appendix provides reference information for the configuration file and includes the following sections:

- Sample rbw.config File
- Description of File Elements
- Summary of Configuration Parameters

Sample rbw.config File

The **rbw.config** file contains the following types of information:

- Site-specific configuration information based on answers to questions asked during the installation procedure
- Option parameters that affect database server behavior
- Tuning parameters that affect database server performance
- Logging parameters that affect logging activities
- Password parameters that set rules for user passwords
- Logical database names and database locations

Many of these parameters can also be set with SQL SET statements or with TMU control statements.

The following example shows **rbw.config** files on UNIX and Windows for Red Brick Decision Server, also known as Red Brick Warehouse. Although your file will vary from the one shown here, this example is typical of a newly installed file. This file is initially built by the installation script, but the database administrator can modify it. Values indicated as <*value*> will be replaced in the **rbw.config** file at your site by the actual value. For Windows, replace / with \ and **/tmp** with **c:\temp**.

For information on modifying the configuration file, refer to "Modifying the Configuration File" on page 9-52. For descriptions of each file element, refer to "Description of File Elements" on page B-9.

UNIX

WIN NT/2000

UNIX

WIN NT/2000

UNIX

```
#############################################################################
#
###
# (C) Copyright 1991-2000 Informix Software, Inc. All rights reserved.
#
# Red Brick Decision Server Version 6.10.0
#
#
# This file contains various parameters for the Red Brick Decision Server
# Daemon (rbwapid), Server (rbwsvr), and Table Management Utilities
# (rb_tmu and rb_ptmu)
#◆
# This file contains various parameters for the Red Brick Warehouse
Service.
# (rbwapid, rbwadmd, rbwlogd, rbwsvr threads), and Table Management Utility
# (rb_tmu)
#◆
#
# The notation {ON | OFF} means use either ON, or OFF, etc.
# The default is the first option shown
#
# This config file was created by user: <redbrick>
#############################################################################
#
###
# The following is used for IPC key values. Note that for shared memory
# and semaphores, the key values will range from the IPC key number to
# IPC key + MAX_SERVERS.
<RB_HOST_NAME> SHMEM 100

# The following is used to specify the communication port for connections
# to the Red Brick Decision Server rbwapid daemon.
<RB_HOST_NAME> SERVER <host>:<port_number>
◆
# The following value controls the maximum number of concurrent Red Brick
# Warehouse sessions that can be started by the rbwapid daemon.
RBWAPI MAX_SERVERS $num_users

# Daemon Start-up user exit script for cleaning up spill files.
RBWAPI CLEANUP_SCRIPT <redbrick_dir>/bin/rb_sample.cleanup

# Unified Logon for Windows NT
#RBWAPI UNIFIED_LOGON {OFF | ON}
◆

# The following value can be used to specify the location of the Red Brick
# Decision Server server image
RBWAPI SERVER_NAME <redbrick_dir>/bin/rbwsvr
◆

# The following value indicates the maximum size of the logfile.
# Once the daemon writes this many lines into the file, it will rename
# it to rbwapid.log_old and a new logfile will be started.
RBWAPI LOGFILE_SIZE 1000

# Process checker daemon checking interval (secs)
```

UNIX

```
#RBWAPI PROCESS_CHECKING_INTERVAL 5

# The maximum number of active databases
#RBWAPI MAX_ACTIVE_DATABASES 30

# Message base details (location and language)
NLS_LOCALE MESSAGE_DIR <redbrick_dir>/message_dir
NLS_LOCALE LOCALE <redbrick_dir>/locale
# First day of the week. 1=Sunday, 2=Monday, ... 7=Saturday
# Use this to override the default first day of the week for the location.
#NLS_LOCALE FIRST_DAYOFWEEK <1-7>

# Server Monitor daemon sampling interval (secs)
RBWMON INTERVAL 120
◆
# Automatic referenced table row generation
#OPTION AUTOROWGEN {OFF | ON}

# Divide by zero control
#OPTION ARITHABORT {ON | OFF}

# Deadlock control
#OPTION ALLOW_POSSIBLE_DEADLOCK {OFF | ON}

# Query Priority control. Uses schema locks if set to OFF.
#OPTION READER_PRIORITY {ON | OFF}

# Enable cross join
#OPTION CROSS_JOIN   {OFF | ON }

# Datatype for COUNT function
#OPTION COUNT_RESULT {INTEGER | DECIMAL | DEC | INT}

# Check index/table report file permissions and location
#OPTION CHECK_REPORT_FILE_PERMISSIONS {SERVER_OWNER | SERVER_GROUP | ALL}
#OPTION CHECK_TABLE_INDEX_DIRECTORY <directory_path>

# Temporary Table creation authorization
#OPTION GRANT_TEMP_RESOURCE_TO_ALL {ON | OFF}

# Generate log records for Advisor?
#OPTION ADVISOR_LOGGING {ON | ON_WITH_CORR_SUB | OFF}

# Assume all precomputed views have been accessed uniformly?
#OPTION UNIFORM_PROBABILITY_FOR_ADVISOR {OFF | ON}

# Automatically rewrite queries using available precomputed views?
#OPTION PRECOMPUTED_VIEW_QUERY_REWRITE {ON | OFF}

# Use invalid precomputed views when rewriting queries?
#OPTION USE_INVALID_PRECOMPUTED_VIEWS {OFF | ON}

# Start DML statement as versioning transaction
#OPTION VERSIONING {OFF | ON}

# The isolation level when accquiring locks for DML statement
#OPTION TRANSACTION_ISOLATION_LEVEL {SERIALIZABLE | REPEATABLE_READ}
```

```
# Start TMU operation as versioning transaction
#OPTION TMU_VERSIONING {OFF | ON | RECOVER}

# Defaults for TMU periodic commit
#OPTION TMU_COMMIT_RECORD_INTERVAL { <ROWCOUNT> | OFF }
#OPTION TMU_COMMIT_TIME_INTERVAL { <SECONDS> | OFF }

# EXPORT default directory
#OPTION EXPORT_DEFAULT_PATH /tmp

# EXPORT field delimiter
#OPTION EXPORT_DELIMITER |

# Bytes per data file for EXPORT command
#OPTION EXPORT_MAX_FILE_SIZE 1K

# Idle timeout for connections in minutes, 0 means feature is disabled
# OPTION IDLE_TIMEOUT 0

############################################################
# Precomputed View Maintenance section
############################################################
# Turn automatic precomputed view maintenance ON or OFF
#OPTION PRECOMPUTED_VIEW_MAINTENANCE {ON | OFF}

# Controls the transaction behavior during precomputed view maintenance.
# This applies only with versioning enabled.
#OPTION PRECOMPUTED_VIEW_MAINTENANCE_ON_ERROR {ROLLBACK | INVALIDATE}

############################################################
#
#  TUNE section: Optional tuning & performance parameters
#
############################################################
# Number of TMU Buffer cache pages
#TUNE TMU_BUFFERS 128

# Tuning parameter for parallel query
#TUNE ROWS_PER_SCAN_TASK 2147483647
#TUNE ROWS_PER_FETCH_TASK 2147483647
#TUNE ROWS_PER_JOIN_TASK 2147483647
#TUNE QUERYPROCS 0
#TUNE TOTALQUERYPROCS 0

#TUNE FORCE_SCAN_TASKS {OFF | <num_tasks>}
#TUNE FORCE_FETCH_TASKS {OFF | <num_tasks>}
#TUNE FORCE_JOIN_TASKS {OFF | <num_tasks>}

# Define physical disk groups -- the default is
# each PSU is in its own file group
#TUNE FILE_GROUP 1 <path1>
#TUNE FILE_GROUP 1 <path2>
#TUNE FILE_GROUP 2 <path3>

# Maximum amount of parallelism to use on a specific file group
#TUNE GROUP 1 1
#TUNE GROUP 2 1
```

```
# Tuning parameter for parallel hybrid hash joins
#TUNE FORCE_HASHJOIN_TASKS {OFF | <num_tasks>}

# Enable parallelism for hybrid hash joins
#TUNE PARALLEL_HASHJOIN  {ON | OFF}

# Tuning parameter for parallel aggregation partitioned by GROUP BY columns
#TUNE FORCE_AGGREGATION_TASKS {OFF | <num_tasks>}

# Enable parallelism for aggregation partitioned by the GROUP BY columns
#TUNE PARTITIONED_PARALLEL_AGGREGATION {ON | OFF}

# Result buffer configuration
#TUNE RESULT_BUFFER {UNLIMITED | <value>{K|M|G}}
#TUNE RESULT_BUFFER_FULL_ACTION {PAUSE | ABORT}

# Index Fillfactor parameters
#FILLFACTOR PI 100
#FILLFACTOR STAR 100
#FILLFACTOR SI 100

#VARCHAR fillfactor parameters
#FILLFACTOR VARCHAR 10

# Optimized index build parameter
#OPTION TMU_OPTIMIZE {OFF | ON}

# Index Temporary Space parameters
# Always specify THRESHOLD before MAXFILESIZE in this
# configuration file
# Specify multiple DIRECTORYs by having multiple
# TUNE INDEX_TEMPSPACE_DIRECTORY entries
#TUNE INDEX_TEMPSPACE_THRESHOLD 10M
#TUNE INDEX_TEMPSPACE_MAXSPILLSIZE 1G
#TUNE INDEX_TEMPSPACE_DIRECTORY /tmp

# Query Temporary Space parameters
# Always specify QUERY_MEMORY_LIMIT before
# QUERY_TEMPSPACE_MAXSPILLSIZE in this configuration file
# Specify multiple DIRECTORYs by having multiple
# TUNE QUERY_TEMPSPACE_DIRECTORY entries
#TUNE QUERY_MEMORY_LIMIT 50M
#TUNE QUERY_TEMPSPACE_MAXSPILLSIZE 1G
#TUNE QUERY_TEMPSPACE_DIRECTORY /tmp

# TMU Row Messages Mode
#TUNE TMU_ROW_MESSAGES {FULL | NONE}

# Default PTMU behavior
#TUNE TMU_SERIAL_MODE { ON | OFF }

# Defaults for PTMU number of tasks used for different operations
#TUNE TMU_CONVERSION_TASKS <no_of_conversion_tasks>
#TUNE TMU_INDEX_TASKS <max_no_index_tasks>
#TUNE TMU_INPUT_TASKS <no_index_tasks>
#TUNE TMU_MAX_TASKS <max_no_tmu_tasks>

# Set limit on TMU memory map space
#TUNE TMU_MMAP_LIMIT <number>{K|M}
```

```
# Random Sampling margin (percentile) to be used for inexact random
sampling
#TUNE RANDOM_SAMPLE_MARGIN <zero_or_positive float>

# Sample size (percentile) to be used by advisor for candidate view
analysis.
#TUNE ADVISOR_SAMPLE_SIZE <0.0 to 99.99>

############################################################
#
#  DEFAULT section
#
############################################################
# Max # of rows to return on an unconstrained query (
#DEFAULT ROWCOUNT 0

# Max # of INFORMATION & STATISTICS messages to return
# for one operation.
#DEFAULT INFO_MESSAGE_LIMIT 1000

# Max # of rows pertaining to LOAD information to be
# retained by RB_DEFAULT_LOADINFO
#DEFAULT RBW_LOADINFO_LIMIT 256

############################################################
#
#  SEGMENTS section
#
# Do NOT set default_data_segment or default_index_segment
# if you have multiple databases.  See the Warehouse
# Administrator's Guide for additional information
#
############################################################
# Segment default directories
#OPTION DEFAULT_DATA_SEGMENT <RB_PATH>
#OPTION DEFAULT_INDEX_SEGMENT <RB_PATH>

# Temporary table default segment directories
#OPTION TEMPORARY_DATA_SEGMENT <RB_PATH>
#OPTION TEMPORARY_INDEX_SEGMENT <RB_PATH>

# Keep/drop segments upon DROP table or index
#OPTION SEGMENTS {KEEP | DROP}

# Segment partial availability controls
#OPTION IGNORE_PARTIAL_INDEXES {OFF | ON}
#OPTION PARTIAL_AVAILABILITY {PRECHECK | INFO | WARN | ERROR}

# Optical storage availability controls
#OPTION IGNORE_OPTICAL_INDEXES {OFF | ON}
#OPTION OPTICAL_AVAILABILITY {WAIT_NONE | WAIT_INFO | WAIT_WARN |
                              SKIP_INFO | SKIP_WARN | ERROR | PRECHECK }

# Default segment expansion controls
#OPTION DEFAULT_SEGMENT_SIZE <number>{K|M|G}
#OPTION DEFAULT_PSU_EXTENDSIZE <number>{K|M|G}
```

```
############################################################
#
#  ADMIN section
#
############################################################
#ADMIN   ACCOUNTING      { OFF | ON }
#ADMIN   ACCT_DIRECTORY  <RB_CONFIG>/logs
#ADMIN   ACCT_MAXSIZE    0
#ADMIN   ACCT_LEVEL      { JOB | WORKLOAD }
#ADMIN   LOGGING         { ON | OFF }
#ADMIN   LOG_DIRECTORY   <RB_CONFIG>/logs
#ADMIN   LOG_MAXSIZE     0
#ADMIN   LOG_AUDIT_LEVEL { ALERT | ROUTINE | URGENT }
#ADMIN   LOG_ERROR_LEVEL { ROUTINE | ALERT | URGENT }
#ADMIN   LOG_OPERATIONAL_LEVEL { ALERT | ROUTINE | URGENT }
#ADMIN   LOG_SCHEMA_LEVEL { ROUTINE | ALERT | URGENT }
#ADMIN   LOG_USAGE_LEVEL { ALERT | ROUTINE | URGENT }
#ADMIN   REPORT_INTERVAL 1
#ADMIN   RENICE_COMMAND  <full_path_of_a_renice_script>
```

UNIX

◆

```
# Create Advisor log files at system startup, and log advisor records?
#ADMIN ADVISOR_LOGGING {OFF | ON}

# Advisor logging directory
#ADMIN ADVISOR_LOG_DIRECTORY <RB_CONFIG>/logs

# Advisor log maximum size control
#ADMIN ADVISOR_LOG_MAXSIZE 0

############################################################
#
#  PASSWORD section
#
############################################################
# Number of days allowed between password changes
#PASSWORD EXPIRATION_DAYS 0
# Number of days before password expires that user will
# begin to be warned that password is about to expire
#PASSWORD EXPIRATION_WARNING_DAYS 0
# Minimum number of days that must pass between
# password changes
#PASSWORD CHANGE_MINIMUM_DAYS 0
# Number of previously used passwords on each account
# against which new passwords will be compared for
# uniqueness
#PASSWORD RESTRICT_PREVIOUS 0
# The following three parameters control the requirements
# for complex passwords. These parameters specify the
# number of characters of the three types that must be
# present in each new password.
#PASSWORD COMPLEX_NUM_ALPHA 0
#PASSWORD COMPLEX_NUM_NUMERICS 0
#PASSWORD COMPLEX_NUM_PUNCTUATION 0
# Minimum required length for new passwords
#PASSWORD MINIMUM_LENGTH 0
# Number of failed login attempts that will result in a
# user account being locked
```

```
#PASSWORD LOCK_FAILED_ATTEMPTS 0
# Number of hours an account will remain locked
#PASSWORD LOCK_PERIOD_HOURS 0
```

WIN NT/2000

```
############################################################
#
#  NETWORK section:  Add additional entries as services
#                    are created
#
############################################################

<RB_HOST_NAME> SERVER <host>:<port_number>

♦

############################################################
#
#  DATABASE section:  Add additional entries as databases
#                     are created
#
############################################################
# Logical database name mappings
DB AROMA <redbrick_dir>/aroma_dir
DB ADMIN <redbrick_dir>/admin_dir
♦
```

UNIX

WIN NT/2000

```
DB AROMA <redbrick_dir>\aroma_db
DB ADMIN <redbrick_dir>\admin_db
♦
```

Description of File Elements

This section provides a brief description of the **rbw.config** file entries present when Red Brick Decision Server is installed with the standard installation procedure.

On Windows, the environment variables **RB_HOST**, **RB_CONFIG**, and **RB_PATH** are defined in the Registry and controlled by the Registry Monitor. On UNIX, they are defined in a startup file.

For instructions on modifying the configuration file after installation, see "Modifying the Configuration File" on page 9-52.

Configuration Information

File Element	Description
<RB_HOST_NAME> SHMEM (UNIX)	Defines the base number for the IPC key range. IPC key values range from SHMEM to SHMEM plus MAX_SERVERS. (Not present on all platforms.) Default: `100` (base 16 integer). This range of numbers must be unique and should be assigned to Red Brick Decision Server by the system administrator or person in charge of maintaining IPC key numbers.
<RB_HOST_NAME> MAPFILE (UNIX)	Specifies the file used as a shared memory map file. (Not present on all platforms.) Default: **./.RB_HOST.mapfile**
<RB_HOST_NAME> SERVER	Defines the host name and port number for all connections made to Red Brick Decision Server. Default: The host name and port number used for the database installation.
RBWAPI MAX_SERVERS	Defines the maximum concurrent connections (users) supported by the warehouse daemon on UNIX or warehouse service on Windows. The number of connections includes 10 for the Web user connection option. Default: 50 (base 10 integer)
RBWAPI CLEANUP_SCRIPT	Defines a spill file cleanup script to be executed on startup of the daemon on UNIX or the warehouse service on Windows. A sample script is shipped with the database server in **bin** subdirectory of the **redbrick** directory. This script is named **rb_sample.cleanup** on UNIX and **rbclean.bat** on Windows.
RBWAPI UNIFIED_LOGON (Windows)	When set to `ON`, the operating system must authenticate each database user, requiring that each user has a corresponding operating system account with privileges to read and write files in the database directories. Default: `OFF`

(1 of 6)

File Element	Description
RBWAPI SERVER_NAME (UNIX)	Specifies the location of the database server image (**rbwsvr**).
RBWAPI LOGFILE_SIZE	Defines the approximate maximum number of lines of the database server logfile, **rbwapid.log**. When this limit is reached, the **rbwapid** daemon process renames the logfile **rbwapid.log_old** and starts a new logfile. However, a multiline message is not split between the old and new log files; **rbwapid.log** can grow beyond the specified limit. Default: `1000`
RBWAPI PROCESS_CHECKING_INTERVAL	Check interval for process checker daemon (**rbwpchk**).
RBWAPI MAX_ACTIVE DATABASES	The maximum number of active databases.
NLS_LOCALE MESSAGE_DIR	Specifies the directory used for the error message files. Default: **./messages** on UNIX or **.\messages** on Windows
NLS_LOCALE LOCALE	Specifies the language, territory, and sort order for the database server. Default: English_UnitedStates.US-ASCII@Binary
NLS_LOCALE FIRST_DAYOF WEEK	Specifies a day as the first day of the week, with 1 representing Sunday and 7 representing Saturday.
RBWMON INTERVAL (UNIX)	Specifies the interval at which the database server monitoring daemon (**rbw.servermon**) checks for database server processes. Default: 120 seconds
OPTION AUTOROWGEN	Turns automatic row generation for referenced tables on or off during TMU load operations. Default: `OFF`
OPTION ARITHABORT	Specifies that arithmetic operations should abort on divide-by-zero errors.

(2 of 6)

File Element	Description
OPTION ALLOW_POSSIBLE_DEADLOCK	Specifies that database servers should wait for a lock, even if a deadlock could result, rather than returning if the possibility of a deadlock exists. Default: not set
OPTION READER_PRIORITY	When set to ON, queries requesting read locks on tables have priority over operations that need to write to the database. When queries are executed, they must acquire a database-level lock on the system tables. When set to OFF, query-priority locking in general is disabled. Schema locks can be acquired at the table level, instead of relying on a database-level lock on the system tables. Default: `ON`
OPTION CROSS_JOIN	Specifies whether cross joins are allowed. Default: `OFF`
OPTION COUNT_RESULT	Specifies INTEGER or DECIMAL data type values for the COUNT function. If tables have more than 2^{32} rows, COUNT_RESULT should be set to DECIMAL. Default: INTEGER
OPTION CHECK_REPORT_FILE_PERMISSIONS	Sets file permissions on the reports generated by the CHECK TABLE and CHECK INDEX statements. The default value, SERVER_OWNER, gives read and write permission to the **redbrick** user. SERVER_GROUP, valid only on UNIX platforms, gives read-only permission to the group. ALL gives read-only permission to all users. Default: SERVER_OWNER
OPTION CHECK_TABLE_INDEX_DIRECTORY	Specifies the directory in which report files generated by CHECK TABLE and CHECK INDEX statements are created. Default: none
OPTION GRANT_TEMP_RESOURCE_TO_ALL	Provides an option to grant or revoke authorization to create temporary tables and indexes to all users with CONNECT system role authorization. Default: `ON`

(3 of 6)

File Element	Description
OPTION ADVISOR_LOGGING	Enables or disables advisor query logging for all sessions. Advisor logging must be enabled, either with the ADMIN ADVISOR_LOGGING ON setting in the **rbw.config** file or with an ALTER SYSTEM START ADVISOR_LOGGING statement, in order for the OPTION ADVISOR_LOGGING statement to take effect. When set to `ON_WITH_CORR_SUB`, correlated subqueries, along with other queries that get rewritten, are logged. When set to `ON`, correlated subqueries are not logged. Only valid with Informix Vista. Possible values: `ON`, `OFF`, `ON_WITH_CORR_SUB` Default: `ON`
OPTION UNIFORM_PROBABILITY_FOR_ADVISOR	Determines whether the Advisor log file is scanned in order to compute the reference count for Advisor system table queries. When set to `ON`, it is assumed that all of the views on a base table are referenced the same number of times. Default: `OFF`
OPTION PRECOMPUTED_VIEW_QUERY_REWRITE	Turns the aggregate query rewrite system on or off. Default: `ON`
OPTION USE_INVALID_PRECOMPUTED_VIEWS	When set to `ON`, uses all precomputed views, including views that are marked invalid (either from SET PRECOMPUTED VIEW *view_name* INVALID commands or from loading or inserting data in the detail table), to rewrite queries with the Vista query rewrite engine. Default: `OFF` On Windows, this parameter also specifies the directory in which spill files for query processing are created. Default: **c:\tmp**
OPTION VERSIONING	Starts DML statements as versioning transactions.
OPTION TRANSACTION_ISOLATION_LEVEL	Sets the isolation level when acquiring locks for DML statements.

(4 of 6)

File Element	Description
OPTION TMU_VERSIONING	Starts TMU operations as versioning transactions. Values are ON, OFF, and RECOVER. The default value is OFF. When set to ON, the vacuum cleaner writes the version log to the target database. This scenario is best for an environment that enables concurrent queries continuously. When set to RECOVER, the loader writes the version log to the target database at every transaction commit. This is a type of blocking load allows higher recoverability for versioned loads. This scenario is best for large nightly loads with exclusive access to the table. This setting is not supported in REORG or offline load operations.
OPTION TMU_COMMIT_RECORD_INTERVAL	Specifies the number of records to load into a table between each commit operation. This statement is valid only with TMU_VERSIONING set to ON or RECOVER. If this statement is used in conjunction with the TMU_COMMIT_TIME_INTERVAL statement, a commit is performed when either condition is met. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: OFF
OPTION TMU_COMMIT_TIME_INTERVAL	Specifies the amount of time, in minutes, to load data into a table before each commit operation.This statement is valid only with TMU_VERSIONING set to ON or RECOVER. If this statement is used in conjunction with the TMU_COMMIT_RECORD_INTERVAL statement, a commit is performed when either condition is met. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: OFF

File Element	Description
OPTION EXPORT_DEFAULT_PATH	Specifies the default directory for files exported using an EXPORT statement. Default: none
OPTION EXPORT_DELIMITER	Specifies the delimiter used in an EXPORT statement that produces an output file in DELIMITED format. Default: |
OPTION EXPORT_MAX_FILE_SIZE	Specifies the maximum file size for files exported using an EXPORT command. Default: none
OPTION IDLE_TIMEOUT	Specifies the number of minutes the server waits before a connection is determined idle and auto-terminated. A setting of `0`, the default, means connections to not time out.
OPTION PRECOMPUTED_VIEW_MAINTENANCE	Specifies if aggregate tables are automatically updated when their parent detail tables are updated. Also called aggregate maintenance. When set to `OFF`, aggregate tables are not automatically updated. Default: `ON`
OPTION PRECOMPUTED_VIEW_MAINTENANCE_ON_ERROR	Specifies error maintenance for a precomputed view in a database using versioning. This option does not apply to databases that are not using versioning. In this case, if errors occur during updates, the precomputed view containing the aggregate table is set to `invalid`. If versioning is in effect, you can specify that updates to the parent table and the aggregate table are rolled back with the `ROLLBACK` setting, or are treated as in non-versioned databases with the `INVALIDATE` setting. Default: `ROLLBACK`

(6 of 6)

Tuning and Performance Parameters

File Element	Description
TUNE TMU_BUFFERS	Specifies the buffer cache size in 8-kilobyte blocks for the TMU. Values range from 128 blocks to 8208 blocks. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: 128 blocks
TUNE ROWS_PER_SCAN_TASK	Specifies the minimum estimated number of rows to be scanned by a relation scan before a parallel relation scan is performed. Default: `2,147,483,647` (Do not enter commas.)
TUNE ROWS_PER_FETCH_TASK	Specifies the minimum estimated number of data rows returned during the fetch portion of STARjoin processing before parallel fetch processes are used. Default: `2,147,483,647` (Do not enter commas.)
TUNE ROWS_PER_JOIN_TASK	Specifies the minimum estimated number of index entries returned during the join processing (index-probing) portion of STARjoin processing before parallel join processes are used. Default: `2,147,483,647` (Do not enter commas.)
TUNE QUERYPROCS	Specifies upper limit on the number of processes used to process a single query. Default: `0`
TUNE TOTALQUERYPROCS	Specifies upper limit on the number of processes used at one time for parallel query processing across all database servers under a single warehouse daemon (in addition to number specified as MAX_SERVERS parameter). Default: `0`
TUNE FORCE_SCAN_TASKS	Specifies the number of parallel processes to use in a relation scan. Default: `OFF`

(1 of 6)

File Element	Description
TUNE FORCE_FETCH_TASKS	Specifies the number of parallel processes to use fetching row data for a single query. Default: `OFF`
TUNE FORCE_JOIN_TASKS	Specifies the number of parallel processes to use processing joins for a single query. Default: `OFF`
TUNE FILE_GROUP	Defines disk groups for purposes of reducing disk seek contention. Default: none
TUNE GROUP	Defines the maximum number of parallel processes per query that access the named disk group concurrently. Default: 1 process per disk group per query
TUNE FORCE_HASHJOIN_TASKS	Specifies the number of parallel processes to use processing hybrid hash joins. Default: `OFF`
TUNE PARALLEL_HASHJOIN	Specifies if parallel processes are used to process hybrid hash joins. Default: `ON`
TUNE FORCE_AGGREGATION_TASKS	Specifies the number of parallel processes to use processing aggregation functions. Default: `OFF`
TUNE PARTITIONED_PARALLEL_AGGREGATION	Specifies if parallel processes are used to process aggregation functions. Default: `OFF`
TUNE RESULT_BUFFER	Specifies the size of the query result buffer. Default: `UNLIMITED`
TUNE RESULT_BUFFER_FULL_ACTION	Specifies whether a query aborts or pauses when the result buffer fills up. Default: `PAUSE`

(2 of 6)

File Element	Description
FILLFACTOR PI, FILLFACTOR STAR, FILLFACTOR SI	Specifies a fill factor percentage to use when creating new index nodes for the primary, STAR, and secondary (user-defined) indexes, respectively. Default: `100`
FILLFACTOR_VARCHAR	Specifies the user-estimated size of a column with VARCHAR data type. Default: `10`
OPTION TMU_OPTIMIZE	Turns optimized index-building on or off for the TMU. Default: `OFF`
TUNE INDEX_TEMPSPACE_THRESHOLD	Specifies the file size (in kilobytes or megabytes) at which spill files for index building are created. Default size: 1 megabyte Default units: kilobytes
TUNE INDEX_TEMPSPACE_MAXSPILLSIZE	Specifies maximum size in kilobytes (K), megabytes (M), or gigabytes (G) to which a file for index building can grow. Default: 1 gigabyte
TUNE INDEX_TEMPSPACE_DIRECTORY	Specifies the directory in which spill files for index building are created. Specify multiple entries for multiple directories with one directory per TUNE INDEX_TEMPSPACE_DIRECTORY parameter. Default: ***/*tmp** on UNIX or ***c:*tmp** on Windows
TUNE QUERY_MEMORY_LIMIT	Specifies the limit to the amount of memory used for query processing in kilobytes (K), megabytes (M), or gigabytes (G), at which spill files for query processing are created. Default size: 50 megabytes Range: 2 megabytes to 4 gigabytes Default: 1 gigabyte

(3 of 6)

File Element	Description
TUNE QUERY_TEMPSPACE_MAXSPILLSIZE	Specifies maximum size in kilobytes (K), megabytes (M), or gigabytes (G) to which a file for query processing can grow.
TUNE QUERY_TEMPSPACE_DIRECTORY	Specifies the directory in which spill files for query processing are created. Specify multiple entries for multiple directories with one directory per TUNE QUERY_TEMPSPACE_DIRECTORY parameter. Default: **/tmp** on UNIX or **c:\tmp** on Windows
TUNE TMU_ROW_MESSAGES	Specifies the warning level for row messages returned during LOAD processing. Default: FULL
TUNE TMU_SERIAL_MODE	Specifies whether the PTMU runs in serial mode in which no parallel processing is used. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: OFF
TUNE TMU_CONVERSION_TASKS	Specifies the number of tasks the PTMU uses to convert input data to the platform-based internal format used to represent data. For more information on this parameter, see the *Table Management Utility Reference Guide*. Default: one-half the number of processors on the computer (as determined from the hardware)

(4 of 6)

File Element	Description
TUNE TMU_INDEX_TASKS	Specifies the number of tasks the PTMU uses to make index entries into nonunique indexes, corresponding to the data being loaded. Each nonunique index can have at most one task associated with it. The number specified with this parameter is the maximum number of tasks that can be used to process all nonunique indexes. The actual number of tasks used is the smaller of the number of nonunique indexes and the number specified with this parameter. For more information on this parameter, see the *Table Management Utility Reference Guide*. Do not use this parameter with the TUNE TMU_MAX_TASKS parameter. Default: one task per nonunique index
TUNE TMU_INPUT_TASKS	Specifies the number of tasks allocated to scan the target table during a REORG operation. Because the number of conversion tasks is always identical to the number of INPUT tasks, this option also controls the number of conversion tasks. Do not use this parameter with the TUNE TMU_MAX_TASKS parameter. Default: none
TUNE TMU_MAX_TASKS	Specifies an upper bound on the total number of tasks allocated for input tasks and index builder tasks. The total number of tasks specified by this parameter must be at least two. Default: none
TUNE TMU_MMAP_LIMIT	Specifies the maximum amount of memory available for mapping indexes during a LOAD or REORG operation. Default: none

File Element	Description
TUNE RAMDOM_SAMPLE_MARGIN	Specifies the percentage of variance from the requested size of returned row samples during query sampling. A message is issued if the amount of returned rows is above or below the specified margin. Default: `10`
TUNE ADVISOR_SAMPLE_SIZE	Specifies the percentage sample size used by Vista candidate view analysis. Vista candidate view analysis uses a random sample of the fact table if the sample view is not specified. A value of `0` or `100` disables the sampling; the full fact table is used for the analysis. Default: `5`

(6 of 6)

Default Parameters

File Element	Description
DEFAULT ROWCOUNT	Specifies the maximum number of rows returned before the database server stops the execution of a query. A value of zero (`0`) turns off the restriction on row retrieval. Default: `0`
DEFAULT INFO_MESSAGE_LIMIT	Specifies the maximum number of informational messages (`STATISTICS` and `INFORMATION`) returned per query. Default: `1000`
DEFAULT RBW_LOADINFO_LIMIT	Specifies the amount of historical load information for all TMU sessions recorded by the system in the **RBW_LOADINFO** system table. Setting this parameter to a value less than 256 causes the **RB_DEFAULT_LOADINFO** file to be truncated but saves the original file as **RB_DEFAULT_LOADINFO.save**. Default: `256`

Segment Parameters

File Element	Description
OPTION DEFAULT_DATA_SEGMENT	Specifies a pathname to a directory in which to place default data segments. (Do not use if database contains multiple databases with default segments.) Default: directory defined by **RB_PATH**
OPTION DEFAULT_INDEX_SEGMENT	Specifies a pathname to a directory in which to place default index segments. (Do not use if database contains multiple databases with default segments.) Default: directory defined by **RB_PATH**
OPTION TEMPORARY_DATA_SEGMENT	Specifies the directory that stores the physical storage units (PSUs) of the default temporary data segments for temporary tables. Default: current database directory
OPTION TEMPORARY_INDEX_SEGMENT	Specifies the directory that stores the PSUs of the default temporary index segments for temporary tables. Default: current database directory
OPTION SEGMENTS	Specifies whether user-defined segments should be dropped or kept when the table or index associated with them is dropped. (Default segments are always dropped.) Default: `KEEP`
OPTION IGNORE_PARTIAL_INDEXES	Specifies that a query should ignore partial indexes and consider only fully available indexes. Default: `OFF`
OPTION PARTIAL_AVAILABILITY	Specifies behavior when a query attempts to access a partially available table or index. Default: `PRECHECK`
OPTION IGNORE_OPTICAL_INDEXES	Specifies whether to use indexes stored on optical segments or not. Default: `OFF`

(1 of 2)

File Element	Description
OPTION OPTICAL_AVAILABILITY	Specifies the query behavior with respect to optical segments. Default: `WAIT_NONE`
OPTION DEFAULT_SEGMENT_SIZE	Specifies the size of default segments for tables and indexes. You can specify the size in kilobytes, megabytes, or gigabytes. If you do not specify a value for DEFAULT_SEGMENT_SIZE, the table size is limited to 2GB (one PSU). If you do not specify a unit for the value, the default unit is kilobytes. When you create tables and indexes stored in default segments, the number of PSUs created is equivalent to the value of DEFAULT_SEGMENT_SIZE divided by 2GB, the INITSIZE is 16K, and the MAXSIZE of each PSU is set to 2GB minus 8k. If you set DEFAULT_SEGMENT_SIZE to a value not evenly divisible by 2GB, the last PSU is smaller than the others.
OPTION DEFAULT_PSU_EXTENDSIZE	Specifies the extend size of PSUs for tables and indexes stored in default segments. The default value is 16K. You can set the extend size in kilobytes, megabytes, or gigabytes. If you do not specify a unit, the default unit is kilobytes. For more information, see OPTION DEFAULT_SEGMENT_SIZE.

(2 of 2)

ADMIN Parameters

File Element	Description
ADMIN ACCOUNTING	Specifies whether the log daemon turns on the accounting feature upon daemon startup. Default: `OFF`
ADMIN ACCT_DIRECTORY	Specifies the location of the file containing the accounting records. Default: **<$RB_CONFIG>/logs** on UNIX or **<%RB_CONFIG%>\logs** on Windows
ADMIN ACCT_MAXSIZE	Sets maximum accounting file size. The minimum value is 10,240 kilobytes.

(1 of 3)

File Element	Description
ADMIN ACCT_LEVEL	Default: limited only by available disk space (Do not enter commas.) Specifies whether basic job accounting or more detailed workload accounting records are captured. Default: `JOB`
ADMIN LOGGING	Specifies whether the log daemon turns on the logging feature upon daemon startup. Default: `ON`
ADMIN LOG_DIRECTORY	Specifies the location of the file containing the logging records. Default: **<$RB_CONFIG>/logs** on UNIX or **<%RB_CONFIG%>\logs** on Windows
ADMIN LOG_MAXSIZE	Sets maximum logging file size. The minimum value is 10 kilobytes. If you do not set this parameter, or if you set it to zero or a negative number, no maximum size is imposed on Advisor log files. Default: limited only by available disk space (Do not enter commas.)
ADMIN LOG_AUDIT_LEVEL	Sets the minimum severity level that is logged for audit events. Default: `ALERT`
ADMIN LOG_ERROR_LEVEL	Sets the minimum severity level that is logged for error events. Default: `ROUTINE`
ADMIN LOG_OPERATIONAL_LEVEL	Sets the minimum severity level that is logged for operational events. Default: `ALERT`
ADMIN LOG_SCHEMA_LEVEL	Sets the minimum severity level that is logged for schema events. Default: `ROUTINE`

File Element	Description
ADMIN LOG_USAGE_LEVEL	Sets the minimum severity level that is logged for usage events. Default: ALERT
ADMIN REPORT_INTERVAL	Sets the maximum interval (in minutes) between dynamic system table refreshes. Default: 1 minute
ADMIN RENICE_COMMAND (UNIX)	Specifies the full pathname of a UNIX **renice** executable file that changes user priorities. Default: none
ADMIN ADVISOR_LOGGING	Determines the startup state of the Advisor log. When this parameter is set to ON, a log file is created when the log daemon (rbwadmd) starts, and log records are captured when aggregate views are used and when candidate views are generated. When this parameter is set to OFF, no log file is created, and data is not logged. Default: OFF
ADMIN ADVISOR_LOG_DIRECTORY	Specifies the location of the file containing the Advisor log records. Default: **<$RB_CONFIG>/logs** on UNIX or **<%RB_CONFIG%>\logs** on Windows
ADMIN ADVISOR_LOG_MAXSIZE	Sets maximum Advisor log file size. The minimum value is 10,240 kilobytes. Default: limited only by available disk space (Do not enter commas.)

(3 of 3)

PASSWORD Parameters

File Element	Description
PASSWORD EXPIRATION_DAYS	Sets the number of days for which each user password is valid. Default: unlimited
PASSWORD EXPIRATION_WARNING_DAYS	Sets the number of days prior to password expiration that a user receives a warning message upon each login. Default: none
PASSWORD CHANGE_MINIMUM_DAYS	Sets the minimum number of days that must pass between password changes.
PASSWORD RESTRICT_PREVIOUS	Sets minimum number of unique passwords that must be used before an expired password can be reused. Default: unlimited
PASSWORD COMPLEX_NUM_ALPHA	Specifies minimum number of alphabetic characters that must be used in each new password. Default: `0`
PASSWORD COMPLEX_NUM_NUMERICS	Specifies minimum number of numeric characters that must be used in each new password. Default: `0`
PASSWORD COMPLEX_NUM_PUNCTUATION	Specifies minimum number of punctuation characters that must be used in each new password. Default: `0`
PASSWORD MINIMUM_LENGTH	Specifies minimum number of total characters that must be used in each new password. Default: `0`
PASSWORD LOCK_FAILED_ATTEMPTS	Sets maximum number of failed database access attempts before a user is locked out of the database. Default: `0`
PASSWORD LOCK_PERIOD_HOURS	Sets number of hours a locked account remains locked. Default: `0`

Database Entries

Entry	Description
DB AROMA	Specifies the mapping between the AROMA logical database name and its physical location. If the Aroma database is installed, this line is present in the configuration file. Default on UNIX: **DB AROMA <*redbrick_dir*>/aroma_db** Default on Windows: **DB AROMA <*redbrick_dir*>\aroma_db**
DB ADMIN	Specifies the mapping between the ADMIN logical database name and its physical location. Default on UNIX: **DB ADMIN <*redbrick_dir*>/admin_db** Default on Windows: **DB ADMIN <*redbrick_dir*>\admin_db**

Summary of Configuration Parameters

The following table lists the various parameters that affect the database server environment and defines how they can be set and what processes they affect. They are listed in the same order they appear in the **rbw.config** file, followed by those parameters that are controlled only by a SET command.

	Set with			Affects		
Parameters	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
RB_HOST SHMEM (UNIX)	✓			✓	✓	rbwapid
RB_HOST MAPFILE (UNIX)	✓			✓	✓	rbwapid
RB_HOST SERVER	✓					rbwapid
RBMON parameter						
RBMON INTERVAL (UNIX)	✓					servermon

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
RBWAPI parameters						
MAX_SERVERS	✓					rbwapid
CLEANUP_SCRIPT	✓					rbwapid
SERVER_NAME (UNIX)	✓					rbwapid
UNIFIED_LOGON (Windows)	✓					rbwapid
LOGFILE_SIZE	✓					rbwapid
PROCESS_CHECKING_ INTERVAL	✓					rbwapid
MAX_ACTIVE_ DATABASES	✓					rbwapid
NLS_LOCALE parameters						
MESSAGE_DIR	✓			✓	✓	all
LOCALE	✓			✓	✓	all
FIRST_DAYOFWEEK	✓			✓	✓	all
TUNE parameters						
TMU_BUFFERS	✓		✓		✓	
ROWS_PER_SCAN_ TASK	✓	✓		✓		
ROWS_PER_FETCH_ TASK	✓	✓		✓		
ROWS_PER_JOIN_ TASK	✓	✓		✓		
QUERYPROCS	✓	✓		✓		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
TOTALQUERYPROCS	✓			✓		rbwapid
FORCE_SCAN_TASK	✓	✓		✓		
FORCE_FETCH_TASK	✓	✓		✓		
FORCE_JOIN_TASK	✓	✓		✓		
FILE_GROUP	✓			✓		rbwapid
GROUP	✓			✓		rbwapid
FORCE_HASHJOIN_ TASK	✓	✓		✓		
PARALLEL_ HASHJOIN	✓	✓		✓		
FORCE_ AGGREGATION_ TASKS	✓	✓		✓		
PARTITIONED_ PARALLEL_ AGGREGATION	✓	✓		✓		
RESULT_BUFFER	✓	✓		✓		
RESULT_BUFFER_ FULL_ACTION	✓	✓		✓		
FILLFACTOR SI, PI, STAR	✓			✓	✓	
FILLFACTOR VARCHAR	✓			✓	✓	
INDEX_TEMPSPACE_ THRESHOLD, MAXSPILLSIZE, DIRECTORY	✓	✓		✓		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
QUERY_MEMORY_LIMIT, QUERY_TEMPSPACE_MAXSPILLSIZE, DIRECTORY	✓	✓		✓		
TMU_ROW_MESSAGES	✓		✓		✓	
TMU_SERIAL_MODE	✓		✓		✓	
TMU_CONVERSION_TASKS	✓		✓		✓	
TMU_INDEX_TASKS	✓		✓		✓	
TMU_INPUT_TASKS	✓		✓		✓	
TMU_MAX_TASKS	✓		✓		✓	
TMU_MMAP_LIMIT	✓		✓		✓	
RANDOM_SAMPLE_MARGIN	✓	✓		✓		
ADVISOR_SAMPLE_SIZE	✓	✓		✓		
OPTION parameters						
AUTOROWGEN	✓		✓		✓	
ARITHABORT	✓	✓		✓		
ALLOW_POSSIBLE_DEADLOCK	✓	✓		✓		
READER_PRIORITY	✓			✓		
CROSS_JOIN	✓	✓		✓		
COUNT_RESULT	✓	✓		✓		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
CHECK_REPORT_ FILE_PERMISSIONS	✓			✓		
CHECK_TABLE_INDEX _DIRECTORY	✓	✓		✓		
GRANT_TEMP_ RESOURCE_TO_ALL	✓			✓		
ADVISOR_LOGGING	✓	✓		✓		
UNIFORM_ PROBABILITY_FOR_ ADVISOR	✓	✓		✓		
PRECOMPUTED_ VIEW_QUERY_ REWRITE	✓	✓		✓		
USE_INVALID_ PRECOMPUTED_ VIEWS	✓	✓		✓		
VERSIONING	✓	✓		✓	✓	
TRANSACTION_ ISOLATION_LEVEL	✓	✓		✓	✓	
TMU_VERSIONING	✓					
TMU_COMMIT_ RECORD_INTERVAL	✓		✓		✓	
TMU_COMMIT_TIME_ INTERVAL	✓		✓		✓	
EXPORT_DEFAULT_ PATH	✓	✓		✓		
EXPORT_DELIMITER	✓			✓		

(5 of 9)

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
EXPORT_MAX_FILE_SIZE	✓	✓		✓		
IDLE_TIMEOUT	✓			✓		
PRECOMPUTED_VIEW_MAINTENANCE	✓	✓		✓	✓	
PRECOMPUTED_VIEW_MAINTENANCE_ON_ERROR	✓	✓		✓	✓	
TMU_OPTIMIZE	✓				✓	
DEFAULT_DATA_SEGMENT	✓	✓		✓		
DEFAULT_INDEX_SEGMENT	✓	✓		✓		
TEMPORARY_DATA_SEGMENT	✓	✓		✓		
TEMPORARY_INDEX_SEGMENT	✓	✓		✓		
SEGMENTS (DROP, KEEP)	✓	✓		✓		
IGNORE_PARTIAL_INDEXES	✓	✓		✓		
PARTIAL_AVAILABILITY	✓	✓		✓		
IGNORE_OPTICAL_INDEXES	✓	✓		✓		
OPTICAL_AVAILABILITY	✓	✓		✓		

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
DEFAULT_ SEGMENT_SIZE	✓	✓		✓		
DEFAULT_PSU_ EXTENDSIZE	✓	✓		✓		
DEFAULT parameters						
ROWCOUNT	✓	✓		✓		
INFO_MESSAGE_ LIMIT	✓			✓		
RBW_LOADINFO_ LIMIT	✓				✓	
ADMIN parameters						
* On UNIX, you can set these parameters with an SQL ALTER SYSTEM statement.						
ACCOUNTING *	✓					rbwlogd
ACCT_DIRECTORY	✓					rbwlogd
ACCT_LEVEL*	✓					rbwlogd
ACCT_MAXSIZE *	✓					rbwlogd
ADVISOR_ LOGGING *	✓			✓		rbwlogd
ADVISOR_LOG_ DIRECTORY	✓			✓		rbwlogd
ADVISOR_LOG_ MAXSIZE	✓			✓		rbwlogd
LOGGING*	✓					rbwlogd
LOG_AUDIT_LEVEL *	✓					rbwlogd
LOG_DIRECTORY	✓					rbwlogd

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
LOG_ERROR_LEVEL *	✓					rbwlogd
LOG_MAXSIZE *	✓					rbwlogd
LOG_OPERATIONAL_ LEVEL *	✓					rbwlogd
LOG_SCHEMA_ LEVEL *	✓					rbwlogd
LOG_USAGE_LEVEL*	✓					rbwlogd
REPORT_INTERVAL	✓	✓		✓	✓	
RENICE_COMMAND (UNIX)	✓			✓		
PASSWORD parameters						
EXPIRATION_DAYS	✓			✓	✓	
EXPIRATION_ WARNING_DAYS	✓			✓	✓	
CHANGE_MINIMUM_ DAYS	✓			✓		
RESTRICT_PREVIOUS	✓					
COMPLEX_NUM_ ALPHA	✓			✓		
COMPLEX_NUM_ NUMERICS	✓			✓	✓	
COMPLEX_NUM_ PUNCTUATION	✓			✓	✓	
MINIMUM_LENGTH	✓			✓	✓	
LOCK_FAILED_ ATTEMPTS	✓			✓	✓	

Parameters	Set with			Affects		
	rbw.config	SQL SET	TMU command	Server	TMU	Daemon
LOCK_PERIOD_ HOURS	✓			✓	✓	
Other parameters						
DB logical database names	✓			✓	✓	all
LOCK (WAIT, NO WAIT)		✓		✓		
			✓		✓	
LOCK (table, database)		✓		✓	✓	
ORDER_BY_ASC_ NULL, ORDER_BY_ DESC_NULL		✓		✓		

* On UNIX, you can set these parameters with an SQL ALTER SYSTEM statement.

(9 of 9)

Appendix

C

System Tables and Dynamic Statistic Tables

This appendix describes the system catalog, the system tables, and the dynamic statistic tables (DSTs) for Red Brick Decision Server. The column name, data type, and a description of the column is provided for each table.

This appendix contains the following sections:

- System Catalog
- Dynamic Statistic Tables
- Data Types and Their Sizes

System Catalog

Red Brick Decision Server maintains system tables that describe the data stored in user databases. You can view contents of the system tables with a SELECT statement, which can include system-table joins. However, the data in the system tables is read-only. It cannot be inserted, updated, or deleted.

The following statement displays a list of tables to which the current user has access. System tables and user-created tables available to the user are displayed.

```
select * from rbw_tables ;
```

The system catalog contains a list of system tables. Each system table contains information for the entire database. When users display information stored in a system table, they see only tables and information about those tables that they created or have permission to access. The **RBW_TABAUTH** and **RBW_USERAUTH** system tables define the tables that each user creates and has permission to access.

Table Name	Table Description
RBW_COLUMNS	Describes the columns from all objects in **RBW_TABLES**.
RBW_CONSTRAINTS	Records the names of primary key and foreign key constraints defined in CREATE TABLE statements.
RBW_CONSTRAINTCOLUMNS	Records the names of columns that make up the primary key and foreign key constraints defined in CREATE TABLE statements.
RBW_HIERARCHIES	Shows the relationships within defined hierarchies.
RBW_INDEXCOLUMNS	Records the columns that form the key of the indexes listed in **RBW_INDEXES**.
RBW_INDEXES	Describes the indexes on all tables listed in **RBW_TABLES**.
RBW_LOADINFO	Provides statistics on recent load operations.
RBW_MACROS	Describes all macros defined in the database.
RBW_OPTIONS	Displays current settings for database parameters that can be tuned.
RBW_PRECOMPVIEW_CANDIDATES	Advisor system table for candidate views (Refer to the *Informix Vista User's Guide*.)
RBW_PRECOMPVIEW_UTILIZATION	Advisor system table for views that are defined in the database (Refer to the *Informix Vista User's Guide*.)
RBW_PRECOMPVIEWCOLUMNS	Shows the relationships between columns in aggregate tables and precomputed views.
RBW_RELATIONSHIPS	Lists tables that share primary foreign key relationships and shows the constraint names applied to those relationships.
RBW_ROLE_MEMBERS	Describes the relationship of all user-created roles to the users and roles that have been granted to them.

(1 of 2)

Table Name	Table Description
RBW_ROLES	Describes all user-created roles defined in the database.
RBW_SEGMENTS	Describes all segments in the system. Only users with the DBA or RESOURCE system role or ACCESS_ANY authorization can access this table.
RBW_STORAGE	Describes the physical storage units (PSUs) used in the system. Only users with the DBA or RESOURCE system role or ACCESS_ANY authorization can access this table.
RBW_SYNONYMS	Describes all table synonyms for tables listed in **RBW_TABLES**.
RBW_TABAUTH	Describes the access rights granted on objects in **RBW_TABLES**.
RBW_TABLES	Describes all tables (including system tables), views, and synonyms in the database.
RBW_USERAUTH	Describes access rights granted to users authorized to use the database.
RBW_VIEWS	Describes the views in **RBW_TABLES** and provides information about precomputed views.
RBW_VIEWTEXT	Describes the text of all views in **RBW_VIEWS**.

(2 of 2)

RBW_COLUMNS Table

The **RBW_COLUMNS** table describes the columns from all database objects listed in the **RBW_TABLES** table. When users display information stored in the **RBW_COLUMNS** table, they see only those columns in objects that they created or have permission to access. This table is updated by the ALTER, CREATE and DROP TABLE, VIEW, and SYNONYM statements. It contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of column.
TNAME	CHAR(128)	Name of table.
SEQ	SMALLINT	Column sequence number.
TYPE	CHAR(12)	Column data type.
LENGTH	SMALLINT	Actual length of column in bytes.
PRECISION	SMALLINT	Specified or implied numeric precision. For TIME and TIMESTAMP columns, the digits display fractional seconds.
SCALE	SMALLINT	Specified scaling factor.
START	INTEGER	The start value of a SERIAL column definition.
STEP	INTEGER	The step value definition of a SERIAL column.
NULLS	CHAR(1)	Flag indicating whether nulls are allowed (`Y` or `N`).
UNIQ	CHAR(1)	Flag indicating whether column values are unique (`Y` or `N`).
PKSEQ	SMALLINT	Sequence of column in primary key; 0 if not a column in primary key.
TID	SMALLINT	Table identifier.
DEFAULTVALUE	CHAR(256)	Default for column.

(1 of 2)

Column Name	Column Type	Column Description
SEGSEQ	SMALLINT	Sequence of segmenting column in segment range specification (`1` for segmenting column, `0` for nonsegmenting columns, `NULL` for views).
FILLFACTOR	SMALLINT	Estimated size of a VARCHAR column expressed as a percentage of the maximum length. Default value is `10`.
USAGE	CHAR(16)	Indicates what the column is used for.
COMMENT	CHAR(256)	User-specified comment; `NULL` if not set with the ALTER TABLE, ALTER SYNONYM, or ALTER VIEW statements.

(2 of 2)

RBW_CONSTRAINTCOLUMNS Table

The **RBW_CONSTRAINTCOLUMNS** table identifies the columns on which primary and foreign key constraints are defined in CREATE TABLE statements.

Column Name	Column Type	Column Description
CONSTRAINT_ NAME	CHAR(128)	The name of the constraint.
TNAME	CHAR(128)	The name of the table containing the constraint.
CNAME	CHAR(128)	The name of the column on which the constraint is defined.
COLSEQ	INTEGER	The sequence of the column in the constraint definition.

RBW_CONSTRAINTS Table

The **RBW_CONSTRAINTS** table describes the primary and foreign key constraints defined in CREATE TABLE statements.

Column Name	Column Type	Column Description
NAME	CHAR(128)	The name of the constraint.
ID	INTEGER	The internal identification number of the constraint.
TYPE	CHAR(11)	The type of constraint: `PRIMARY KEY` or `FOREIGN KEY`.
TNAME	CHAR(128)	The name of the table containing the constraint.
CREATOR	CHAR(128)	The user who created or last altered the table definition.

RBW_HIERARCHIES Table

The **RBW_HIERARCHIES** table shows the table and column relationships within a hierarchy.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of the hierarchy.
FROM_TABLE	CHAR(128)	Table from which values are mapped.
FROM_COLUMN	CHAR(128)	Column from which values are mapped.
TO_ TABLE	CHAR(128)	Table to which values are mapped.
TO_COLUMN	CHAR(128)	Column to which values are mapped.
CONSTRAINT_NAME	CHAR(128)	Names the foreign key constraint through which a rollup relationship is defined. Indicates `NULL` if the rollup relationship is within the same table.

RBW_INDEXCOLUMNS Table

The **RBW_INDEXCOLUMNS** table describes the index keys on all indexes listed in the **RBW_INDEXES** table. This table stores one row for each column in an index key.

When users display information stored in the **RBW_INDEXES** table, they see only indexes of objects they created or have permission to access. This table is updated by CREATE TABLE, DROP TABLE, CREATE INDEX, or DROP INDEX statements and contains the following columns.

Column Name	Column Type	Column Description
INAME	CHAR(128)	Name of index.
TNAME	CHAR(128)	Name of table.
CNAME	CHAR(128)	Name of column in key.
SEQ	SMALLINT	Sequence number of column in key.
FKNAME	CHAR(128)	Name of the foreign key constraint for STAR indexes. Returns NULL for non-STAR indexes.

RBW_INDEXES Table

The **RBW_INDEXES** table describes the indexes on all objects listed in the **RBW_TABLES** table. When users display information stored in the **RBW_INDEXES** table, they see only indexes on columns in objects they created or have permission to access. This table is updated by CREATE TABLE, DROP TABLE, ALTER INDEX, CREATE INDEX, or DROP INDEX statements and contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of index. Default primary key indexes are named <***table_name***>**_PK_IDX**.
TNAME	CHAR(128)	Name of table.
TYPE	CHAR(7)	Index type: BTREE, STAR, TARGET, TARGETS, TARGETM, or TARGETL.

(1 of 2)

Column Name	Column Type	Column Description
CNAME	CHAR(128)	Indexed column name; first column name on multi-column indexes.
CREATOR	CHAR(128)	Creator of index.
DATETIME	TIMESTAMP	Date and time of index creation. NULL indicates index is under construction.
FILLFACTOR	INTEGER	Fill factor setting of index.
INTACT	CHAR(1)	Flag indicating whether index is intact (Y), or there is detected but unrepaired damage (N).
PARTIAL	CHAR(1)	Whether table is partially available due to one or more offline segments (Y or N).
STATE	CHAR(20)	VALID, INVALID, BUILDING.
COMMENT	CHAR(256)	User-specified comment; NULL if not set with the ALTER INDEX statement.

(2 of 2)

RBW_LOADINFO Table

The **RBW_LOADINFO** table describes data loads into tables and offline segments. This table is updated by LOAD DATA statements and contains one row for each load operation. Only the most recent 256 rows are retained. Older rows are deleted automatically. This table contains the following columns.

Column Name	Column Type	Column Description
TNAME	CHAR(128)	Name of table.
SEGNAME	CHAR(128)	Name of segment if load performed into an offline segment; NULL if not an offline load.
USERNAME	CHAR(128)	Name of user that performed load.
STARTED	TIMESTAMP	Date and time load started.

(1 of 3)

Column Name	Column Type	Column Description
FINISHED	TIMESTAMP	Date and time load completed.
MODE	CHAR(20)	Mode used to insert or modify rows (`INSERT`, `REPLACE`, `APPEND`, `MODIFY`, `MODIFY AGGREGATE`, `UPDATE`, `UPDATE AGGREGATE`).
STATUS	CHAR(128)	Success: Load completed without error. Incomplete: Load encountered a nonfatal error and can roll back the table to a consistent state and indicate progress with a message. Error: Load fails, either in initial stages before modifying table, or later when it cannot roll table back to consistent state.
INSERTED	DECIMAL(18)	Number of rows inserted into table.
UPDATED	DECIMAL(18)	Number of existing rows in the table that were updated.
SKIPPED	DECIMAL(18)	Number of rows in the input file that were skipped, as specified by a START RECORD clause.
DISCARDED	DECIMAL(18)	Number of rows in the input file that were rejected and discarded.
AUTOROWGEN	CHAR(1)	Whether any rows were automatically generated (`Y` or `N`).
COMMENT	CHAR(256)	User-specified comment or descriptive data; `NULL` if not specified in the LOAD DATA statement.

(2 of 3)

Column Name	Column Type	Column Description
TRANSACTION_TYPE	CHAR (32)	Transaction type for the command. Possible values: READ_ONLY, VERSIONING, BLOCKING.
READ_REVISION	DECIMAL (10,0)	The revision number of the database being accessed by the transaction.
NEW_REVISION	DECIMAL (10,0)	The revision number of the database the transaction creates. NULL for aborted or uncommitted transactions.

(3 of 3)

RBW_MACROS Table

The **RBW_MACROS** table describes all macros in the database. When a member of the DBA system role or a user with ACCESS_ANY authorization displays information stored in the **RBW_MACROS** table, the user can see all public and private macros. All other users can see only those macros the user created or those defined as PUBLIC. Only the creator of a temporary macro can see information on that temporary macro. This table is updated by CREATE and DROP MACRO statements and contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of macro.
TYPE	CHAR(9)	Macro type: PUBLIC, PRIVATE, or TEMPORARY.
NARGS	SMALLINT	Number of macro arguments expected.
CREATOR	CHAR(128)	Creator of macro.
DATETIME	TIMESTAMP	Date and time of macro creation.

(1 of 2)

Column Name	Column Type	Column Description
CATEGORY	SMALLINT	Syntax category for macro. Values less than 256 can be defined by Red Brick Decision Server; NULL if not set with the CREATE MACRO statement. For information about categories that Red Brick Decision Server defines, refer to the *SQL Reference Guide*.
COMMENT	CHAR(256)	User-specified comment or descriptive data; NULL if not set with the CREATE MACRO or ALTER MACRO statement.
TEXT	CHAR(1024)	Text of macro definition.

(2 of 2)

RBW_OPTIONS Table

The **RBW_OPTIONS** table lists the values of all tunable parameters in the database. This table is updated when a user issues a **SET** command during the current session. It contains the following columns.

Column Name	Column Type	Column Description
USERNAME	CHAR(128)	Name of user running the current session.
OPTION_NAME	CHAR(128)	Name of parameter.
VALUE	CHAR(1024)	Current value of parameter.

RBW_PRECOMPVIEW_CANDIDATES Table

For a description of the **RBW_PRECOMPVIEW_CANDIDATES** Advisor system table, refer to the *Informix Vista User's Guide*.

RBW_PRECOMPVIEW_UTILIZATION Table

For a description of the **RBW_PRECOMPVIEW_UTILIZATION** Advisor system table, refer to the *Informix Vista User's Guide*.

RBW_PRECOMPVIEWCOLUMNS Table

The **RBW_PRECOMPVIEWCOLUMNS** table shows the relationship between columns in aggregate tables and precomputed views.

Column Name	Column Type	Column Description
TNAME	CHAR(128)	Name of the aggregate table associated with the precomputed view.
TCOLUMN	CHAR(128)	Name of the column in the aggregate table.
VNAME	CHAR	Name of the precomputed view.
VCOLUMN	CHAR	Name of the column in the precomputed view.

RBW_RELATIONSHIPS Table

The **RBW_RELATIONSHIPS** table describes the primary key / foreign key relationships between tables in a schema. It contains the following columns.

Column Name	Column Type	Column Description
PKTABLE	CHAR(128)	The name of the referenced (dimension) table.
FKTABLE	CHAR(128)	The name of the referencing (fact) table.
PKCONSTRAINT	CHAR(128)	The name of the primary key constraint in the referenced table. If the CREATE TABLE statement does not name the constraint, a name is generated by appending the string _PKEY_CONSTRAINT to the name of the table.

(1 of 2)

Column Name	Column Type	Column Description
FKCONSTRAINT	CHAR(128)	The name of the foreign key constraint as specified by the referencing table. If the CREATE TABLE statement does not name the constraint, a name is generated by appending the string _FKEY*N*_CONSTRAINT to the name of the table, where *N* is a number that identifies the ordinal position of the foreign key specification, as defined by the referencing table.
DELACTION	CHAR(9)	Action triggered by DELETE: `CASCADE`, `NO_ACTION`.
CREATOR	CHAR(128)	The user who created the table.

(2 of 2)

RBW_ROLE_MEMBERS Table

The **RBW_ROLE_MEMBERS** table describes the relationship of all user-created roles and their members (all users and roles that have been granted the role). When a user displays information stored in the table **RBW_ROLE_MEMBERS**, the user sees all user-created roles that have members. This table is updated by GRANT and REVOKE statements.

Column Name	Column Type	Column Description
ROLENAME	CHAR(128)	Name of role.
USERNAME	CHAR(128)	Name of user or user-created role that has been granted ROLENAME.
INDIRECT	CHAR(1)	Whether the user or user-created role is an indirect member of ROLENAME (`Y` if an indirect member; `N` if a direct member).
ADDED	TIMESTAMP	Date and time the user or user-created role became a member of ROLENAME.

RBW_ROLES Table

The **RBW_ROLES** table describes the user-created roles in the database. When a user displays information stored in the **RBW_ROLES** table, the user sees all user-created roles in the database.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of role.
CREATOR	CHAR(128)	Creator of role.
CREATED	TIMESTAMP	Date and time role was created.
COMMENT	CHAR(256)	User-specified comment; NULL if not set with the ALTER ROLE statement.

RBW_SEGMENTS Table

The **RBW_SEGMENTS** table describes all segments in the system. When a member of the DBA or RESOURCE system role or a user with ACCESS_ANY authorization displays information in the **RBW_SEGMENTS** table, the user sees all segments. When a member of the CONNECT system role displays this table, the user sees no segments. This table is updated by ALTER SEGMENT, CREATE SEGMENT, DROP SEGMENT, CREATE TABLE, DROP TABLE, CREATE INDEX, DROP INDEX, BACKUP, RESTORE, and LOAD DATA statements. It contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of segment.
TNAME	CHAR(128)	Name of table that uses segment for row data or indexes; set to NULL for unattached segments.
CREATOR	CHAR(128)	Creator of segment.
DATETIME	TIMESTAMP	Date and time of segment creation.
NPSUS	INTEGER	Number of physical storage units (PSUs) used for segment.

(1 of 3)

Column Name	Column Type	Column Description
NCOLS	INTEGER	*Number of columns used to segment the data or index.*
MINKEY	CHAR(256)	Minimum key value in the segment; displays first 256 characters.
MAXKEY	CHAR(256)	Maximum key value in the segment; displays first 256 characters.
ID	INTEGER	Segment ID.
TOTALFREE	INTEGER	Kilobytes of unused space in segment. Refer to "TOTALFREE Column" on page 9-22
INAME	CHAR(128)	Name of index that uses segment; set to NULL for unattached and row data segments.
ONLINE	CHAR(1)	Whether a segment is online (Y or N). For system segment (NULL).
OPTICAL	CHAR(1)	Whether a segment contains one or more optical PSUs (Y or N). (NULL for system segment.)
INTACT	CHAR(1)	Flag indicating whether segment is intact (Y), or there is detected, unrepaired damage (N). For system segment (NULL).
INSYNCH	CHAR(1)	Whether the row contents are synchronized with the indexes of table. For offline segments: Y or N; for online segments: Y; for index segments, unattached segments, and system segment: NULL.
LAST_OFFLINE	TIMESTAMP	Date and time the segment last set offline; initially contains segment creation time.
LAST_ONLINE	TIMESTAMP	Date and time segment last set online; initially contains segment creation time.
LAST_LOAD	TIMESTAMP	Completion time of the last offline load into segment; initially set to NULL.

(2 of 3)

Column Name	Column Type	Column Description
COMMENT	CHAR(256)	User-specified comment or descriptive data; NULL if not set with the ALTER SEGMENT statement.
LOCAL_ID	SMALLINT	The segment ID that is returned in the **RBW_SEGID** pseudocolumn.
USAGE	CHAR (32)	The current function that a segment is performing. Possible values: UNUSED, TABLE, INDEX, LOAD_WORK, LOAD_INDEX, BACKUP_DATA, or VERSION_LOG.
ROWCOUNT	DECIMAL(10)	The number of rows in the segment.

(3 of 3)

RBW_STORAGE Table

The **RBW_STORAGE** table describes the PSUs in the system. When a member of the DBA or RESOURCE system role or a user with ACCESS_ANY authorization displays information in the **RBW_STORAGE** table, the user sees all PSUs. When a member of the CONNECT system role displays this table, the user sees no PSUs. This table is updated by CREATE SEGMENT, DROP SEGMENT, CREATE TABLE, DROP TABLE, CREATE INDEX, DROP INDEX, and BACKUP statements and contains the following columns.

Column Name	Column Type	Column Description
SEGNAME	CHAR(128)	Segment name containing PSU.
SEGID	SMALLINT	Segment ID containing PSU.
PSEQ	INTEGER	Sequence number of PSU in segment.
LOCATION	CHAR(1024)	The location of PSU.
MAXSIZE	INTEGER	Kilobytes of maximum allowed size of PSU. Refer to "MAXSIZE Column" on page 9-20, "Adding Space to a Segment" on page 9-26, and "Changing PSU Sizes" on page 9-33.

(1 of 2)

Column Name	Column Type	Column Description
INITSIZE	INTEGER	Kilobytes of initial allocated size of PSU. Refer to "Changing PSU Sizes" on page 9-33.
EXTENDSIZE	INTEGER	Kilobytes of increment size to use when extending PSU. Refer to "Changing PSU Sizes" on page 9-33.
USED	INTEGER	Kilobytes of current area in use in PSU. Refer to "USED Column" on page 9-21.
INTACT	CHAR(1)	Flag indicating whether PSU is intact (Y), or there is detected, unrepaired damage (N). For system segment (NULL).
PHYSICAL_LOCATION	CHAR (1024)	Actual location of the PSU.

(2 of 2)

RBW_SYNONYMS Table

The **RBW_SYNONYMS** table describes all table synonyms for tables in the database. When users display information stored in the **RBW_SYNONYMS** table, they see only those synonyms they created or have permission to access. This table is updated by ALTER, CREATE and DROP SYNONYM statements and contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of synonym.
TNAME	CHAR(128)	Name of table referenced by synonym.
CREATOR	CHAR(128)	Creator of synonym.
COMMENT	CHAR(256)	User-specified comment; NULL if not set with the ALTER SYNONYM statement.

RBW_TABAUTH Table

The **RBW_TABAUTH** table describes the object privileges granted on tables. When users display information stored in the **RBW_TABAUTH** table, they see only object privileges for tables they created or have permission to access. This table is updated by GRANT and REVOKE statements and contains the following columns.

Column Name	Column Type	Column Description
GRANTEE	CHAR(128)	User or role granted object privilege to a table or view. (Object privileges can be granted to user-created roles.)
GRANTOR	CHAR(128)	User granting privilege to GRANTEE.
TNAME	CHAR(128)	Name of table, view, or synonym.
SELAUTH	CHAR(1)	Whether grantee has SELECT privilege (Y, N, R, or I).
INSAUTH	CHAR(1)	Whether grantee has INSERT privilege (Y, N, R, or I).
DELAUTH	CHAR(1)	Whether grantee has DELETE privilege (Y, N, R, or I).
UPDAUTH	CHAR(1)	Whether grantee has UPDATE privilege (Y, N, R, or I).
DATETIME	TIMESTAMP	Date and time object privilege granted.

The authorizations are as follows.

Value	Description
Y	User has the task authorization.
N	User does not have the object privilege.
R	User has the object privilege directly through a role.
I	User has the object privilege indirectly through a role.

RBW_TABLES Table

The **RBW_TABLES** table lists all tables, views, and synonyms in the database. When a user displays information stored in the **RBW_TABLES** table, the user sees only those objects that the user created or has permission to access. This table is updated by CREATE and DROP TABLE, VIEW, or SYNONYM statements and contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of table, view, or synonym.
TYPE	CHAR(8)	Type of object: `TABLE`, `VIEW`, `SYNONYM`, or `SYSTEM`.
CREATOR	CHAR(128)	Creator of table (blank for system tables).
ID	SMALLINT	Table identifier. Negative numbers identify system tables.
DATETIME	TIMESTAMP	Date and time of table creation.
MAXSEGMENTS	INTEGER	Maximum number of segments allowed for the table (specified with CREATE TABLE or ALTER TABLE); indicates `NULL` if not specified.
MAXROWS_PER_SEG	DECIMAL(10)	Maximum number of rows allowed per segment (specified with CREATE TABLE or ALTER TABLE); indicates `NULL` if not specified.
MAXSIZE_ROWS	DECIMAL(10)	Maximum number of rows (calculated from MAXSIZE parameter of CREATE SEGMENT for all PSUs in each segment attached).
SEGMENT_BY	CHAR(11)	Segmentation scheme of table (range, hash, `NULL`).

(1 of 2)

Column Name	Column Type	Column Description
INTACT	CHAR(1)	Flag indicating whether table is intact (Y), or there is detected, unrepaired damage (N).
PARTIAL	CHAR(1)	Flag indicating whether table is partially available due to one or more offline segments (Y or N).
COMMENT	CHAR(256)	User-specified comment; NULL if not set.

(2 of 2)

RBW_USERAUTH Table

The **RBW_USERAUTH** table describes access rights granted to users. When a member of the DBA system role or a user with ACCESS_ANY authorization displays information in the **RBW_USERAUTH** table, the user sees authorizations for all users. When a member of the RESOURCE or CONNECT system role displays this table, the user sees only the user's own authorization. This table is updated by GRANT and REVOKE statements and contains the following columns. For a description of the authorizations, see "RBW_TABAUTH Table" on page C-18.

Column Name	Column Type	Column Description
GRANTEE	CHAR(128)	User or role granted authorization. Task authorizations can be granted to user-created roles.
GRANTOR	CHAR(128)	User granting authorization to GRANTEE.
DBAAUTH	CHAR(1)	Whether GRANTEE is a member of DBA system role (Y, N, R, or I).
RESAUTH	CHAR(1)	Whether grantee is a member of RESOURCE system role (Y, N, R, or I).
DATETIME	TIMESTAMP	Date and time authorization granted.

(1 of 4)

Column Name	Column Type	Column Description
LOCKED	CHAR(1)	Whether or not GRANTEE's account is locked due to failed connection attempts (Y or N); NULL for roles.
EXPIRED	CHAR(1)	Whether or not the account for GRANTEE is expired because of failure to change password before the defined expiration date (Y or N); NULL for roles.
PASSWORD_TS	TIMESTAMP	Date and time database password for GRANTEE was added or last changed.
USER_MANAGEMENT	CHAR(1)	Whether GRANTEE can alter, create and drop database users and change passwords (Y, N, R, or I).
GRANT_TABLE	CHAR(1)	Whether GRANTEE can grant object privileges to database users and to roles (Y, N, R, or I).
ROLE_MANAGEMENT	CHAR(1)	Whether GRANTEE can alter, create, drop, grant, and revoke roles (Y, N, R, or I).
ALTER_ANY	CHAR(1)	Whether GRANTEE can alter indexes, segments, tables, macros, views, and synonyms (Y, N, R, or I).
PUBLIC_MACROS	CHAR(1)	Whether GRANTEE can create and drop PUBLIC macros (Y, N, R, or I).
ACCESS_ANY	CHAR(1)	Whether GRANTEE can select data from all database objects, including private user information in the system tables (Y, N, R, or I).
MODIFY_ANY	CHAR(1)	Whether GRANTEE can insert, update, delete, and load any data (Y, N, R, or I).

(2 of 4)

Column Name	Column Type	Column Description
DROP_ANY	CHAR(1)	Whether GRANTEE can drop objects created by any user (Y, N, R, or I).
CREATE_ANY	CHAR(1)	Whether GRANTEE can create any object, including those that use another's resources (Y, N, R, or I).
LOCK_DATABASE	CHAR(1)	Whether GRANTEE can lock the database (Y, N, R, or I).
BACKUP_DATABASE	CHAR(1)	Whether GRANTEE can back up the database (Y, N, R, or I).
RESTORE_DATABASE	CHAR(1)	Whether GRANTEE can restore the database (Y, N, R, or I).
UPGRADE_DATABASE	CHAR(1)	Whether GRANTEE can upgrade the database (Y, N, R, or I).
REORG_ANY	CHAR(1)	Whether GRANTEE can reorganize any table or index (Y, N, R, or I).
OFFLINE_LOAD	CHAR(1)	Whether GRANTEE can use any segment as a working segment for offline loads or synchronize segments after offline loads (Y, N, R, or I).
ALTER_SYSTEM	CHAR(1)	Whether GRANTEE can issue the ALTER SYSTEM statement to perform database administration tasks (Y, N, R, or I).
ACCESS_SYSINFO	CHAR(1)	Whether GRANTEE can query the Dynamic Statistic Tables for statistics about database activity (Y, N, R, or I).
ALTER_TABLE_INTO_ANY	CHAR(1)	Whether GRANTEE can alter own tables into segments for another users (Y, N, R, or I).

(3 of 4)

Column Name	Column Type	Column Description
CREATE_OWN	CHAR(1)	Whether GRANTEE can create own objects: indexes, private macros, segments, synonyms, tables, and views (`Y`, `N`, `R`, or `I`).
DROP_OWN	CHAR(1)	Whether GRANTEE can drop own objects (`Y`, `N`, `R`, or `I`).
ALTER_OWN	CHAR(1)	Whether GRANTEE can alter own indexes, macros, segments, synonyms, tables, and views (`Y`, `N`, `R`, or `I`).
GRANT_OWN	CHAR(1)	Whether GRANTEE can grant object privileges on own objects to other users (`Y`, `N`, `R`, or `I`).
IGNORE_QUIESCE	CHAR(1)	Flag indicating whether a user can access a quiesced database (`Y`) or is locked out (`N`).
ACCESS_ADVISOR_INFO	CHAR(1)	Flag indicating whether a user can query the Advisor system tables (`Y`) or not (`N`).
TEMP_RESOURCE	CHAR(1)	Flag indicating whether a user has the authority to create temporary tables (`Y`) or not (`N`).
ISROLE	CHAR(1)	Whether GRANTEE is a role (`Y` if a role; `N` if a user).
PRIORITY	SMALLINT	Value between 0 and 100 indicating user priority for database server resources. `0` is high priority; `100` is low.
EXPORT	CHAR(1)	Whether GRANTEE can issue the EXPORT statement to perform database administration tasks (`Y`, `N`, `R`, or `I`).
COMMENT	CHAR(256)	User-specified comment; `NULL` if not set.

(4 of 4)

RBW_VIEWS Table

The **RBW_VIEWS** table contains the names of all views in the database. When users display information stored in the **RBW_VIEWS** table, they see only those views they created or have permission to access. This table is updated by the ALTER VIEW, CREATE VIEW and DROP VIEW statements and contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of view.
CREATOR	CHAR(128)	Creator of view.
PRECOMPVIEW	CHAR(1)	Denotes whether the view is precomputed (`Y` or `N`).
PRECOMPVIEW_TABLE	CHAR(128)	Name of the aggregate table associated with the precomputed view. Indicates `NULL` if the view is not a precomputed view.
DETAIL_TABLE	CHAR(128)	Denotes the detail table on which the aggregate table is defined. Indicates `NULL` if the view is not a precomputed view.
VALID	CHAR(1)	Indicates whether the data in the aggregate table matches the data in the detail table. Indicates `NULL` if the view is not a precomputed view.
MAINTENANCE	CHAR(7)	Indicates whether aggregate maintenance is performed. Possible values are `NULL` (for non-precomputed views), `ON`, `OFF`, or `REBUILD`.
COMMENT	CHAR(256)	User-specified comment; `NULL` if not set with the ALTER VIEW statement.

RBW_VIEWTEXT Table

The **RBW_VIEWTEXT** table describes the text of all views listed in the **RBW_VIEWS** table. When users display information stored in the **RBW_VIEWTEXT** table, they see only the text of those views they created or have permission to access. If the text of a view is longer than 256 characters, the view text spans multiple rows. This table is updated by CREATE VIEW and DROP VIEW statements and contains the following columns.

Column Name	Column Type	Column Description
NAME	CHAR(128)	Name of view.
SEQ	INTEGER	Sequence number of the view text.
TEXT	CHAR(1024)	Text of view definition (including CREATE VIEW keywords).

Dynamic Statistic Tables

The Dynamic Statistic Tables (DSTs) can help you monitor database activity. The DSTs include:

- **DST_COMMANDS**
- **DST_DATABASES**
- **DST_LOCKS**
- **DST_SESSIONS**
- **DST_USERS**

For more information on the DSTs, refer to "Monitoring Database Activity with Dynamic Statistic Tables" on page 8-6.

DST_COMMANDS Table

The **DST_COMMANDS** table contains information about each command issued against the database by a currently connected session. This information consists of cumulative statistics for each command. Any subprocesses that the command spawns are included in the statistics calculations.

The following table lists and describes the **DST_COMMANDS** columns.

Column Name	Column Type	Description
DBNAME	CHAR(128)	Logical database name.
UNAME	CHAR(128)	User name for the user issuing the command.
NODE_NAME	CHAR(128)	Name of node on which command is running (for MPP servers).
PID	INTEGER	Process ID for session running the command.
STARTED	TIMESTAMP	Start time of the command.
STATE	CHAR(64)	■ Connecting ■ Idle ■ Executing ■ Returned *x* rows; computed *y* rows (for query) ■ Inserted *x* rows (for insert) ■ Deleted *x* rows (for delete) ■ Updated *x* rows (for update)
COMMAND	CHAR(1024)	Starting text of current command prior to macro expansion.
CACHE_READS	INTEGER	Number of times that a block was found in local buffer cache (avoiding a logical read request).
CACHE_WRITES	INTEGER	Number of times that a block was found in local buffer cache (avoiding a logical write request).

(1 of 3)

Column Name	Column Type	Description
LOGICAL_READS	INTEGER	Number of logical reads performed by the command.
LOGICAL_WRITES	INTEGER	Number of logical writes performed by the command.
PHYSICAL_READS	INTEGER	Number of physical reads performed by the command (`NULL` for platforms that do not support this statistic).
PHYSICAL_WRITES	INTEGER	Number of physical writes performed by the command (`NULL` for platforms that do not support this statistic).
SYSTEM_CPUTIME	DEC(9,2)	System CPU time used by the command (in seconds).
USER_CPUTIME	DEC(9,2)	User CPU time used by the command (in seconds).
SPILL_COUNT	INTEGER	The number of spill files used in an operation.
MEMORY_USED	INTEGER	Amount of memory being used by the command (in kilobytes).
TEMPSPACE_USED	INTEGER	Amount of spill space used by the command (in kilobytes).
PARALLELISM	INTEGER	Number of parallel tasks being performed by the command.
TRANSACTION_TYPE	CHAR (32)	Transaction type for the command. Possible values: `READ_ONLY`, `VERSIONING`, `BLOCKING`.
READ_REVISION	DECIMAL (10,0)	The revision number of the database being accessed by the transaction.

(2 of 3)

Column Name	Column Type	Description
NEW_REVISION	DECIMAL (10,0)	The revision number of the database the transaction creates. NULL for aborted or uncommitted transactions.
TRANSACTION_ ISOLATION_LEVEL	CHAR (32)	The isolation level of the transaction. Possible values: REPEATABLE_READ, SERIALIZABLE.
LAST_UPDATED	TIMESTAMP	Time stamp of when this row was last updated.

(3 of 3)

DST_DATABASES Table

The DST_DATABASES table contains statistics that indicate the overall level of activity against a database. It also contains the database location and state (quiescent or active).

The following table lists and describes all the DST_DATABASES columns.

Column Name	Column Type	Description
DBNAME	CHAR(128)	Logical database name.
DBLOCATION	CHAR(1024)	Database directory path.
CURRENT_CONNECTS	INTEGER	Current number of connected sessions.
PEAK_CONNECTS	INTEGER	Maximum number of concurrent sessions.
TOTAL_CONNECTS	INTEGER	Cumulative count of connected sessions.
TOTAL_FATAL_EXITS	INTEGER	Number of times a session has terminated abnormally. This number includes any invalid login attempts and failed attempts against quiesced databases.

(1 of 3)

Column Name	Column Type	Description
TOTAL_COMMANDS	INTEGER	Number of commands executed against this database.
QUIESCED	CHAR(1)	Flag indicating whether the database is quiesced (Y) or active (N).
ADMINDB	CHAR(1)	Flag indicating whether the database is the administration database (Y) or a user-created database (N).
VERSION_LOG_ SEGMENT	CHAR (128)	The name of the segment containing the version log. NULL if version log does not exist.
VERSIONING_ STARTED	CHAR (1)	Whether versioning is currently enabled on the database (Y, N, or NULL if version log does not exist).
ACTIVE_VACUUM_ CLEANERS	INTEGER	The number of vacuum cleaner daemon processes active for the database (0 or 1). NULL if version log does not exist.
CURRENT_REVISION	DECIMAL (10, 0)	The database revision number of the most recently committed version of the database. NULL if version log does not exist.
OLDEST_ACTIVE_ REVISION	DECIMAL (10, 0)	The database revision number of the oldest committed version of the database having at least one active read process. NULL if version log does not exist.
QUERY_REVISION	DECIMAL(10,0)	The database revision number being used by a particular user query. If the query revision is not set, this column is null.

(2 of 3)

Column Name	Column Type	Description
LATEST_MERGED_ REVISION	DECIMAL (10, 0)	The database revision number of the latest version of the database that has been merged from the version log to the main database PSUs. NULL if version log does not exist.
VERSION_LOG_USED	INTEGER	The amount of disk space (in kilobytes) used in the version log for new versions of database blocks. NULL if version log does not exist.
VERSION_LOG_ AVAILABLE	INTEGER	The amount of free disk space (in kilobytes) available in the version log. NULL if version log does not exist.
VERSION_LOG_ MAXIMUM_USED	INTEGER	The maximum amount of disk space (in kilobytes) used in the version log. Also known as the *high water mark*. Can be reset with the ALTER SYSTEM RESET STATISTICS statement. NULL if version log does not exist.
MAXREVISIONS	INTEGER	The maximum number of active revisions allowed in the database. NULL if version log does not exist. Can be changed with an ALTER DATABASE CREATE VERSION LOG IN statement.
LAST_UPDATED	TIMESTAMP	Time stamp of when this row was last updated.

(3 of 3)

DST_LOCKS Table

The **DST_LOCKS** table contains information about the locks that each session is holding or waiting for. If the session is waiting for a lock, the **DST_LOCKS** table gives some information on the process that is holding that lock (blocking).

The following table lists and describes the **DST_LOCKS** columns.

Column Name	Column Type	Description
DBNAME	CHAR(128)	Database logical name.
UNAME	CHAR(128)	User name.
NODE_NAME	CHAR(128)	Node name where the process is running (for MPP systems).
PID	INTEGER	PID of this process.
TNAME	CHAR(128)	Table being locked (`NULL` for segment lock).
SEGNAME	CHAR(128)	Name of the segment for a segment lock (`NULL` for table only lock).
DATETIME	TIMESTAMP	Time of the lock request.
TYPE	CHAR(2)	Lock type used for the current transaction. Possible values: read-only (`RO`), read-key (`RK`), read-data (`RD`), write-blocking (`WB`), write-key (`WK`), and write-data (`WD`).
BLOCKER_UNAME	CHAR(128)	Name of user holding lock or `NULL` if current process is holding the lock.
BLOCKER_PID	INTEGER	PID of process holding lock that is blocking current attempt or `NULL` if the current process holds the lock.
BLOCKER_NODE	CHAR(128)	Node name where the blocking process is running (for MPP systems) or `NULL` if the current process is holding the lock.
LAST_UPDATED	TIMESTAMP	Time stamp of when this row was last updated.

DST_SESSIONS Table

The **DST_SESSIONS** table contains information on each session currently connected to the database. This information includes both *cumulative* statistics over all of the commands issued by the session and *peak* statistics (the maximum single command values).

Column Name	Column Type	Description
DBNAME	CHAR(128)	Logical database name.
UNAME	CHAR(128)	Database user name for user running the session.
PID	INTEGER	The **rbwsvr** process ID for the session.
COMPONENT	CHAR(32)	`SERVER`, `WORKGROUP SERVER`, `TMU`, or `PTMU`.
STARTED	TIMESTAMP	Start time of the session.
NET_ADDRESS	CHAR(32)	Network address of client processes if any
CLIENT_TOOL	CHAR(32)	Name of client front-end tool or `NULL` if no client tool used.
GATEWAY	CHAR(128)	Gateway identifier or `NULL` if no gateway used.
INDEX_TEMPSPACE_ DUPLICATESPILLPERCENT	INTEGER	Percentage of the index building temporary space allocated for duplicates. Only valid for REORG operations.
PRIORITY	SMALLINT	Current priority of this session.
QUERY_MEMORY_LIMIT	INTEGER	Size at which queries are written to disk (in 8-kilobyte blocks).
QUERY_TEMPSPACE_ DIRECTORIES	CHAR(1024)	Directories for query-related temporary space.

(1 of 4)

Column Name	Column Type	Description
QUERY_TEMPSPACE_ MAXSPILLSIZE	INTEGER	Maximum amount of temporary space per query (in 8-kilobyte blocks).
INDEX_TEMPSPACE_ DIRECTORIES	CHAR (1024)	Directories for index-building temporary space.
INDEX_TEMPSPACE_ MAXSPILLSIZE	INTEGER	Maximum amount of temporary space per index-building operation (in 8-kilobyte blocks).
REPORT_INTERVAL	INTEGER	Current session reporting interval (in minutes).
TOTAL_COMMANDS	INTEGER	Number of statements executed during this session.
TOTAL_CANCELS	INTEGER	Number of statements canceled during this session.
TOTAL_CACHE_READS	INTEGER	Number of times that a block was found in the local buffer cache (avoiding a logical read request).
TOTAL_CACHE_WRITES	INTEGER	Number of times that a block was found in the local buffer cache (avoiding logical write requests).
TOTAL_LOGICIAL_READS	INTEGER	Maximum number of logical reads performed by this session.
TOTAL_LOGICAL_WRITES	INTEGER	Number of logical writes performed by this session.
TOTAL_PHYSICAL_READS	INTEGER	Maximum number of physical reads performed by this session (NULL for platforms that do not support this statistic).
TOTAL_PHYSICAL_ WRITES	INTEGER	Number of physical writes performed by this session (NULL for platforms that do not support this statistic).

(2 of 4)

Column Name	Column Type	Description
TOTAL_SYSTEM_CPUTIME	DEC(9,2)	System CPU time used by this session (in seconds).
TOTAL_USER_CPUTIME	DEC(9,2)	User CPU time used by this session (in seconds).
TOTAL_SPILL_COUNT	INTEGER	Number of times a spill area was used by this session.
PEAK_CACHE_READS	INTEGER	Maximum number of times that a block was found in local buffer cache for a single session command (avoiding logical write requests).
PEAK_CACHE_WRITES	INTEGER	Maximum number of times that a block was found in the local buffer cache for a single session (avoiding logical write requests).
PEAK_LOGICIAL_READS	INTEGER	Maximum number of logical reads performed by a command within the current session.
PEAK_LOGICAL_WRITES	INTEGER	Maximum number of logical writes performed by a command within the current session.
PEAK_PHYSICAL_READS	INTEGER	Maximum number of physical reads performed by a command within the current session (NULL for platforms that do not support this statistic).
PEAK_PHYSICAL_WRITES	INTEGER	Maximum number of physical writes performed by a command within the current session (NULL for platforms that do not support this statistic).
PEAK_SYSTEM_CPUTIME	DEC(9,2)	Maximum system CPU time used by this user to access this database within the session (in seconds).

(3 of 4)

Column Name	Column Type	Description
PEAK_USER_CPUTIME	DEC(9,2)	Maximum user CPU time used by a command within the session (in seconds).
PEAK_SPILL_COUNT	INTEGER	Maximum number of times a spill area was used by a command within the current session.
PEAK_PARALLELISM	INTEGER	Maximum number of parallel tasks performed by a command within the session.
PEAK_MEMORY_USED	INTEGER	Maximum memory (in kilobytes) used by a single session command to access this database.
PEAK_TEMPSPACE_USED	INTEGER	Maximum amount of spill space (in kilobytes) used by a single session command to access this database.
LAST_UPDATED	TIMESTAMP	Time stamp of when this row was last updated.

(4 of 4)

DST_USERS Table

The **DST_USERS** table contains information about each user who has accessed the database since the database server was last started. The statistics for each user can be reset with an ALTER SYSTEM RESET STATISTICS statement. This information includes both *cumulative* statistics over all of the sessions for a user on the database and *peak* statistics; that is, maximum single-session values.

The following table lists and describes the **DST_USERS** columns.

Column Name	Column Type	Description
DBNAME	CHAR(128)	Logical database name.
UNAME	CHAR(128)	User name.
FIRST_LOGIN	TIMESTAMP	Time of user's first access of this database.
LAST_LOGIN	TIMESTAMP	Time of user's most recent access of this database.
CURRENT_CONNECTS	INTEGER	Number of currently connected sessions.
PEAK_CONNECTS	INTEGER	Maximum number of concurrent sessions at any time by this user.
TOTAL_CONNECTS	INTEGER	Total number of connects by this user.
TOTAL_FATAL_EXITS	INTEGER	Number of times this user had a session terminate abnormally.
TOTAL_COMMANDS	INTEGER	Total number of commands executed by this user.
TOTAL_CANCELS	INTEGER	Number of canceled commands by this user.
TOTAL_CACHE_READS	INTEGER	Number of times that a block was found in local buffer cache (avoiding a logical read request).
TOTAL_CACHE_WRITES	INTEGER	Number of times that a block was found in local buffer cache (avoiding a logical write request).
TOTAL_LOGICAL_READS	INTEGER	Number of logical reads performed by this user on this database.
TOTAL_LOGICAL_WRITES	INTEGER	Number of logical writes performed by this user on this database.

(1 of 3)

Column Name	Column Type	Description
TOTAL_PHYSICAL_READS	INTEGER	Number of physical reads performed by this user on this database (NULL for platforms that do not support this statistic).
TOTAL_PHYSICAL_WRITES	INTEGER	Number of physical writes performed by this user on this database (NULL for platforms that do not support this statistic).
TOTAL_SYSTEM_CPUTIME	DEC(9,2)	Cumulative system CPU time used by this user to access this database (in seconds).
TOTAL_USER_CPUTIME	DEC(9,2)	Cumulative user CPU time used by this user to access this database (in seconds).
TOTAL_SPILL_COUNT	INTEGER	Cumulative count of the number of times that a spill area was used.
PEAK_CACHE_READS	INTEGER	Maximum number of times that a block was found in the local buffer cache for a single session (avoiding logical read requests).
PEAK_CACHE_WRITES	INTEGER	Maximum number of times that a block was found in the local buffer cache for a single session (avoiding logical write requests).
PEAK_LOGICIAL_READS	INTEGER	Maximum number of logical reads performed by this user on this database in one session.
PEAK_LOGICAL_WRITES	INTEGER	Maximum number of logical writes performed by this user on this database in one session.
PEAK_PHYSICAL_READS	INTEGER	Maximum number of physical reads performed by this user on this database in one session (NULL for platforms that do not support this statistic).

(2 of 3)

Column Name	Column Type	Description
PEAK_PHYSICAL_WRITES	INTEGER	Maximum number of physical writes performed by this user on this database in one session (NULL for platforms that do not support this statistic).
PEAK_SYSTEM_CPUTIME	DEC(9,2)	Maximum system CPU time used by this user to access this database in one session (in seconds).
PEAK_USER_CPUTIME	DEC(9,2)	Maximum user CPU time used by this user to access this database in one session (in seconds).
PEAK_SPILL_COUNT	INTEGER	Maximum number of times that spill area was used by one session.
PEAK_PARALLELISM	INTEGER	Maximum number of parallel tasks performed by this user on this database in one session.
PEAK_MEMORY_USED	INTEGER	Maximum memory (in kilobytes) used by this user to access this database in one session.
PEAK_TEMPSPACE_USED	INTEGER	Maximum amount of spill area (in kilobytes) used by this user in one session.
LAST_UPDATED	TIMESTAMP	Time stamp of when this row was last updated.

(3 of 3)

Data Types and Their Sizes

The following table defines the size in bytes of each data type.

Data Types	Size in Bytes
CHARACTER	Length (number of bytes). Maximum is 1024.
VARCHAR	Length (number of bytes) of the literal or expression that created the value or the fill factor, whichever is greater. Maximum is `1024`. Default is `1`.
DATE	3
TIME	3 without fractional seconds; 5 with fractional seconds
TIMESTAMP	6 without fractional seconds; 8 with fractional seconds
INTEGER	4 (range: -2^{31} to $2^{31}-1$; 2^{31} = 2,147,483,648)
SERIAL	4 (range: 1 to $2^{31}-1$; 2^{31} = 2,147,483,648)
SMALLINT	2 (range: -2^{15} to 2^{15} - 1; 2^{15} = 32,768)
TINYINT	1 (range: -2^{7} to 2^{7} - 1; 2^{7} = 128)
FLOAT, DOUBLE	8 (range: approximately 1.E-308 to 1.E308)
REAL	4 (range: approximately 1.E-38 to 1.E37)

(1 of 2)

Data Types	Size in Bytes
DECIMAL or NUMERIC with precision:	
1-2	1
3-4	2
5-9	4
10-11	5
12-14	6
15-16	7
17-18	8
19-21	9
22-23	10
24-26	11
27-28	12
29-31	13
32-33	14
34-35	15
36-38	16

(2 of 2)

For definitions and use of data types, refer to the *SQL Reference Guide*.

Appendix

D

Example: Using Segments with Time-Cyclic Data

This example illustrates how multiple segments can be used in a time-cyclic database. The example includes a simple star schema with a single STAR index, and the index is segmented in the same way as the data. The example demonstrates two techniques for management of time-cyclic data:

- Rolling off old segments and reusing them with new data.
- Creating new segments and adding them to the database.

This appendix includes the following sections:

- Background
- Rolling Off and Reusing Data and Index Segments
- Adding a New Segment
- Using an Offline Load Operation
- Deleting the Oldest Data
- Reusing the Segments

Background

The database used for this example contains three dimension (referenced) tables, **Period**, **Product**, and **Market**, and a fact (referencing) table, **Sales**, which contains sales data for two full years (eight quarters) plus the current quarter:

- Data for the **Sales** table is divided into quarterly segments, with data for each quarter residing in a separate user-defined segment.
- The **Sales** data for the current quarter is updated daily. At the end of each quarter, data for the oldest quarter is removed from the table, data for the current quarter becomes part of the two-year history, and a new current quarter is started.
- The **Sales** table has a user-created index, a STAR index, that resides in the same number of segments as the **Sales** table. The automatically created B-TREE index on the primary key columns has been dropped because the STAR index serves as the primary key index for the **Sales** table.
- Data and indexes for the dimension tables all reside in default segments.
- In the **Period** table, the primary key (**Perkey**) is the DATE data type.

The following figure illustrates the tables used in this example.

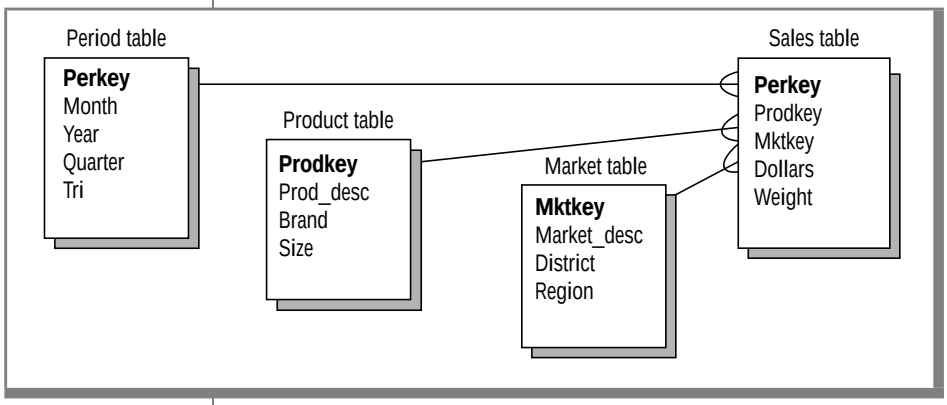


Figure D-1
Schema Example

Assume the database is created during Q1 2001 and loaded with data for eight full quarters (all of 1999 and 2000) plus data for the current quarter, which is updated weekly. Q2 2001 is about to start. You have two tasks:

- Before Q2 starts, you need to add a new segment to hold Q2 2001 data.
- After Q2 starts, you need to remove the Q1 1999 data.



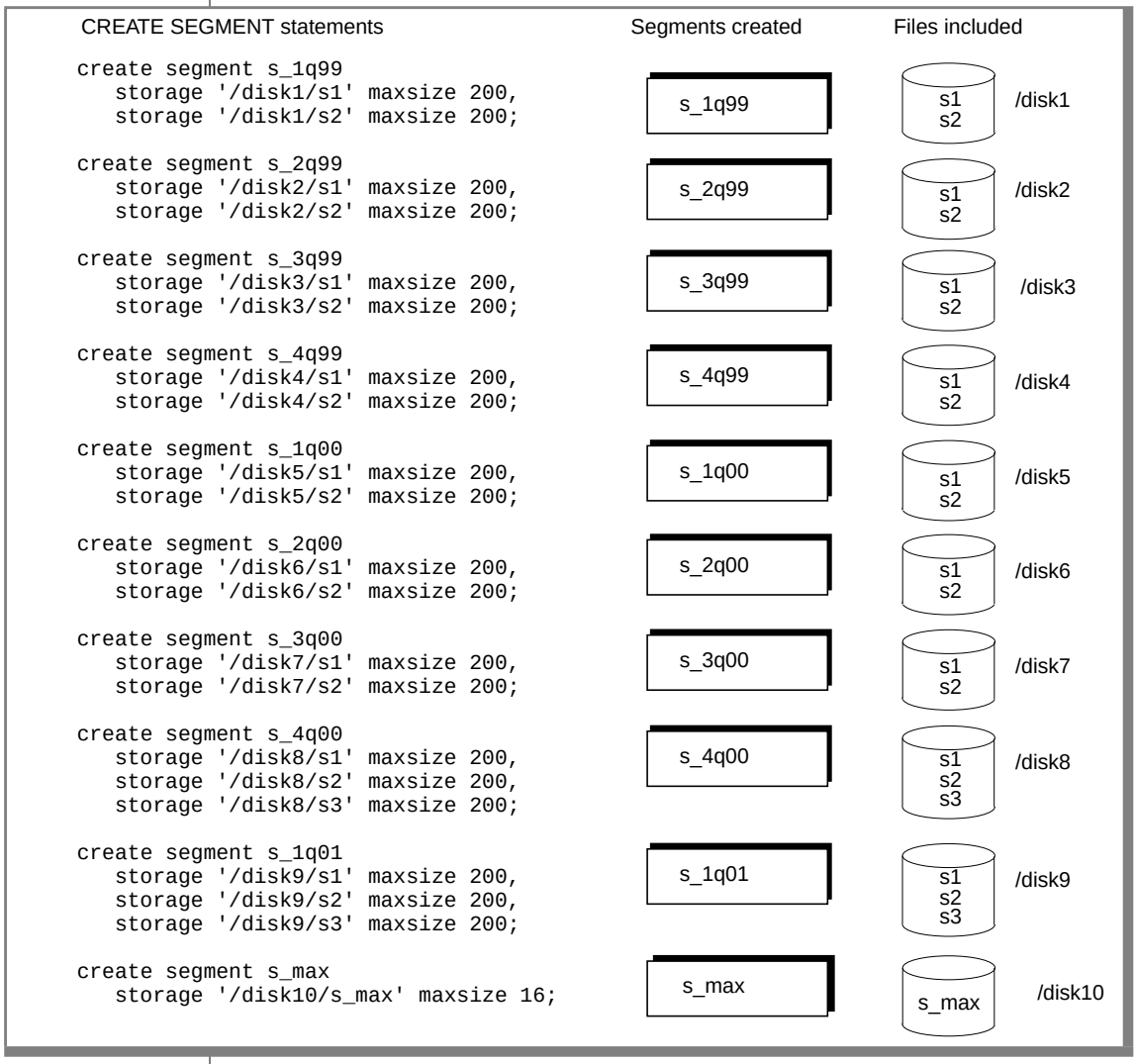
Tip: *When you are ready to delete the data for the oldest quarter, you need to decide whether to save the segment for reuse by detaching it from the table and then reattaching it wherever you want to use it or whether to just drop the segment.*

The Data Segments

UNIX

For the purpose of this example, assume the data segments for the **Sales** table are defined as follows.

Figure D-2
Sales Table Segments on UNIX

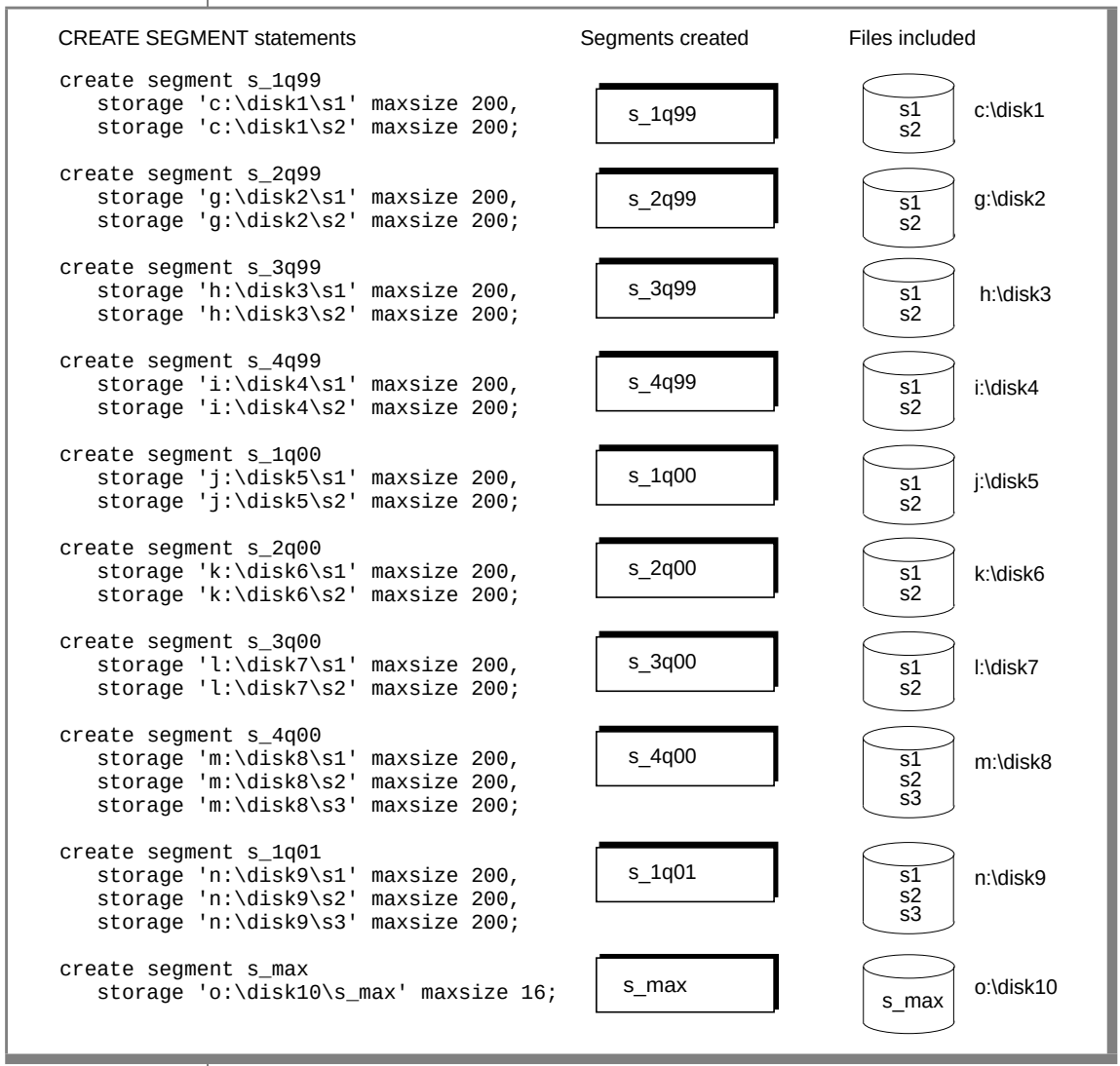


♦

WIN NT/2000

For the purpose of this example, assume the data segments for the **Sales** table are defined as follows.

Figure D-3
Sales Table Segments on Windows



◆

The last segment, **s_max**, is an empty segment that is a placeholder at the high end of the range. This segment remains empty and therefore can be searched more quickly than a segment containing actual data. (When a segment is attached, the surrounding segments are searched to verify that they contain no data that overlaps the new segment range.) While such a segment is not necessary, its existence allows a new segment to be attached more quickly.

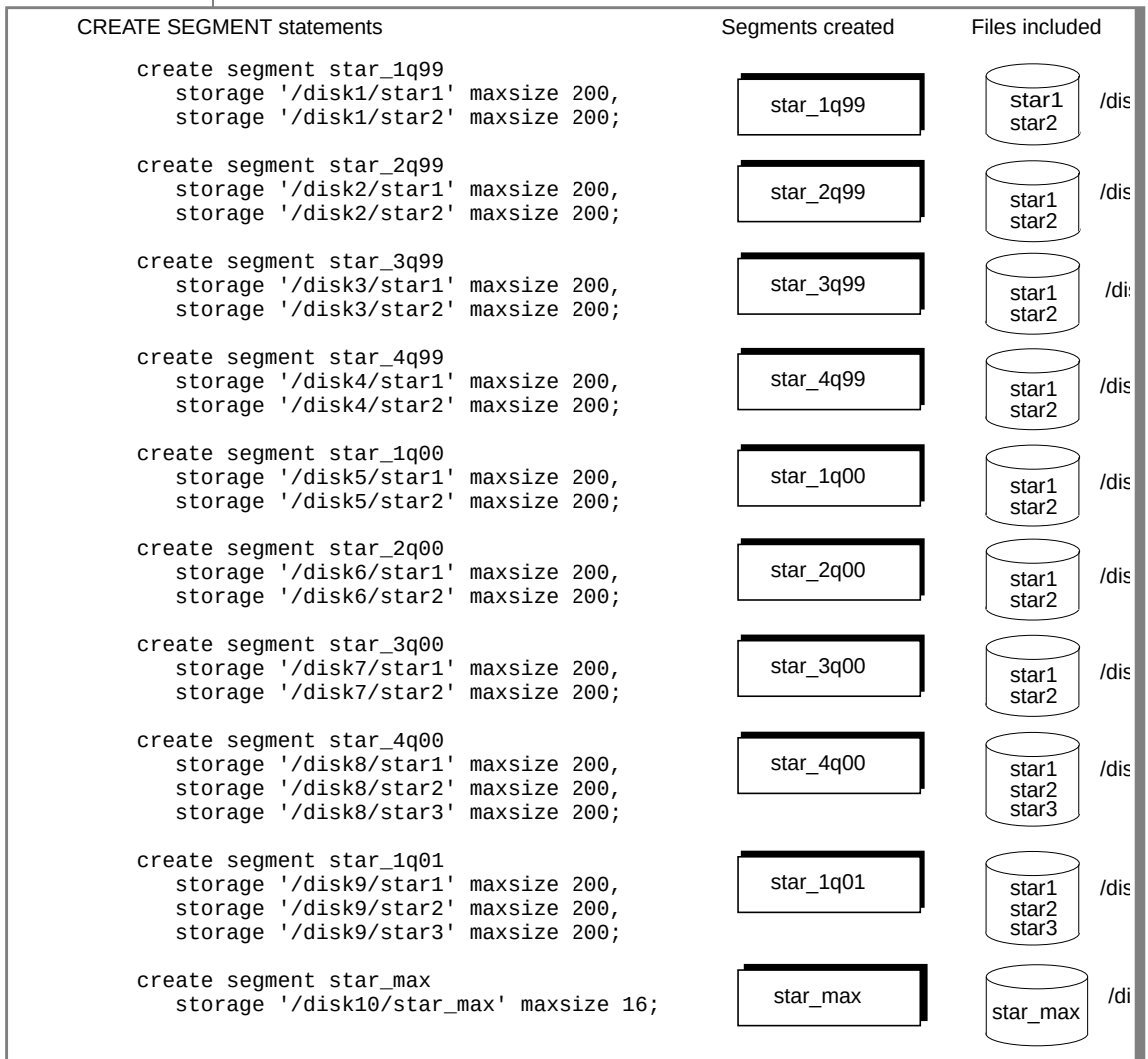
You must create the segments before you create the tables that use them.

The Index Segments

UNIX

For this example, assume the index segments for the **Sales_star** index are defined as follows.

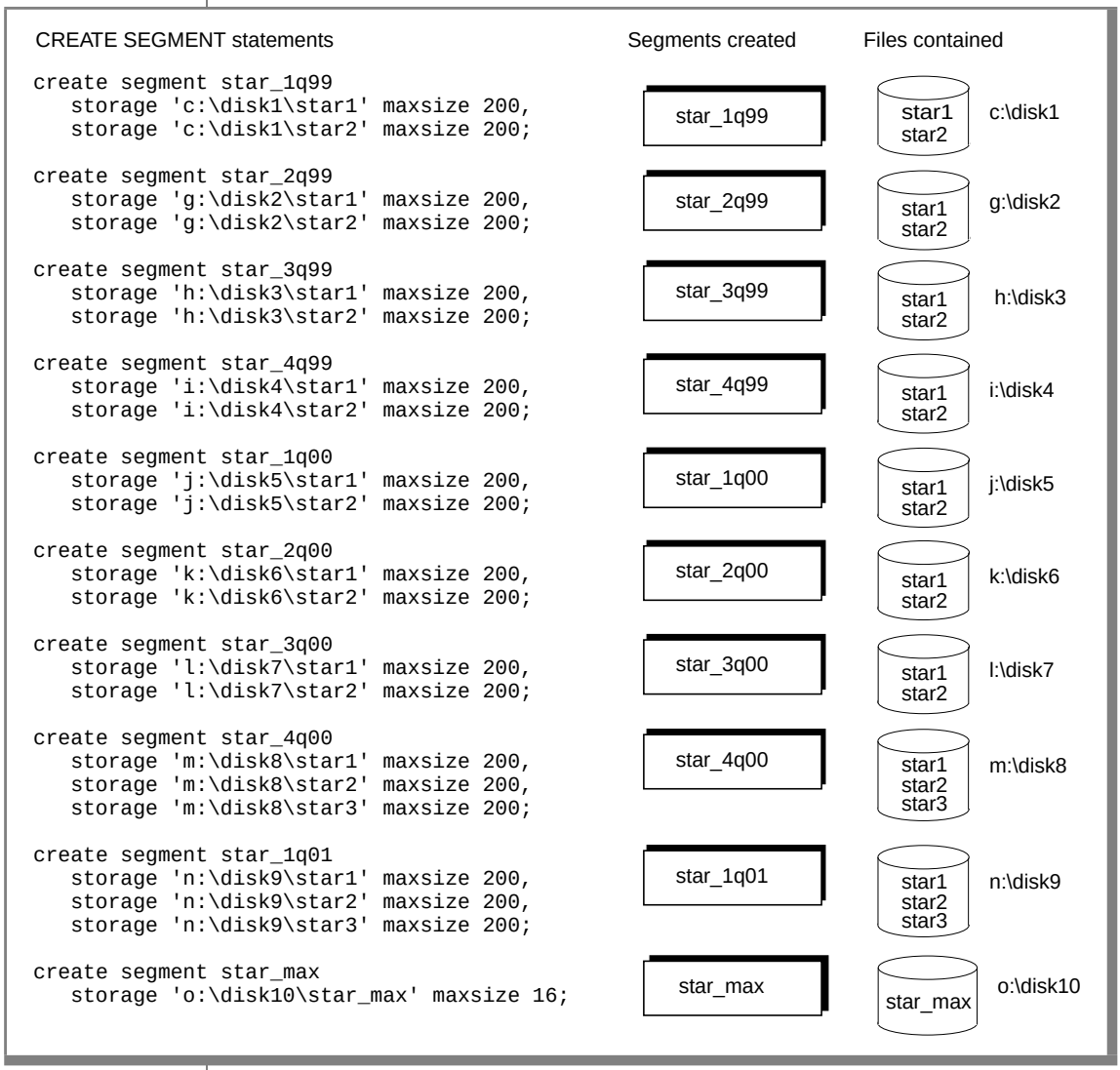
Figure D-4
Sales_star Index on UNIX



WIN NT/2000

For this example, assume the index segments for the **Sales_star** index are defined as follows.

Figure D-5
Sales_star Index on Windows



♦

The last segment, **star_max**, is an empty segment that is a placeholder at the high end of the range. This segment remains empty and therefore can be searched more quickly than a segment containing actual data. (When a segment is attached, the surrounding segments are searched to verify that they contain no data that overlaps the new segment range.) While such a segment is not necessary, its existence allows a new segment to be attached more quickly.

You must create the segments before you create the indexes that use them.

The Tables

Assume the **Period** table resides in a default segment and is defined as follows:

```
create table period (
   perkey date not null,
   month char (15),
   year integer,
   quarter integer,
   tri integer,
   primary key (perkey))
   maxrows per segment 2048;
```

Assume the **Sales** table was created and segmented by quarters with the following CREATE TABLE statement. The segment specification assigns the segments, segmenting column, and ranges of data that can reside in each segment. The **Perkey** column is the segmenting column. Therefore, the ranges indicate values in the **Perkey** column. The range for each segment defines one quarter, so each quarter of sales data resides in a separate segment.

```
create table sales (
   perkey date not null,
   prodkey integer not null,
   mktkey integer not null,
   dollars decimal (7, 2),
   weight smallint,
   primary key (perkey, prodkey, mktkey),
   foreign key (perkey) references period (perkey),
   foreign key (prodkey) references product (prodkey),
   foreign key (mktkey) references market (mktkey))
   data in (s_1q99, s_2q99, s_3q99, s_4q99, s_1q00, s_2q00,
      s_3q00, s_4q00, s_1q01, s_max)
   segment by values of (perkey) ranges (
      min:DATE'1999-04-01',
      DATE'1999-04-01':DATE'1999-07-01',
```

```
        DATE'1999-07-01':DATE'1999-10-01',
        DATE'1999-10-01':DATE'2000-01-01',
        DATE'2000-01-01':DATE'2000-04-01',
        DATE'2000-04-01':DATE'2000-07-01',
        DATE'2000-07-01':DATE'2000-10-01',
        DATE'2000-10-01':DATE'2001-01-01',
        DATE'2001-01-01':DATE'2001-04-01',
        DATE'2001-04-01': max);
```

The STAR Index

Assume a STAR index covering all of the foreign keys is created on the **Sales** table and that the STAR index is segmented *exactly* like the data; that is, the index entries corresponding to each row of data in the **s_1q99** data segment are in the **star_1q99** index segment, the index entries corresponding to each row of data in the **s_2q99** data segment are in the **star_2q99** index segment, and so on. To accomplish this, specify the range for each segment to include only the dates corresponding to a given quarter.

```
create star index sales_star
      on sales (perkey, prodkey, mktkey)
      in (star_1q99, star_2q99, star_3q99, star_4q99,
          star_1q00, star_2q00, star_3q00, star_4q00,
          star_1q00, star_max)
      segment by references of (perkey)
      ranges (min:90, 90:181, 181:273, 273:365, 365:455,
        455:546, 546:638, 638:730, 730:821, 821:max)
```



Tip: *The ranges in the CREATE INDEX statement refer to row numbers in the referenced table* ***Period****, not row numbers in the* ***Sales*** *table. For the complete syntax of the CREATE INDEX statement, refer to the "SQL Reference Guide."*

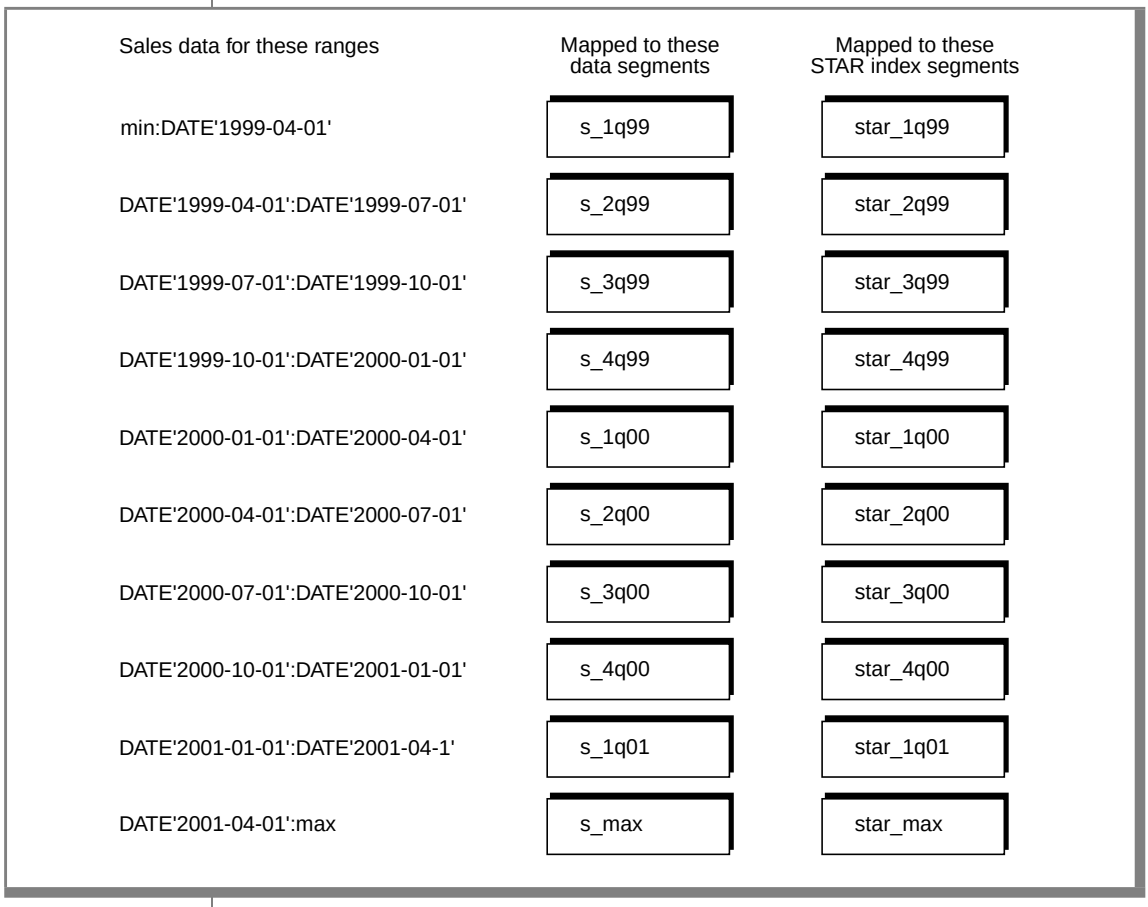
To find the boundaries for the STAR index segment ranges, query the **RBW_ROWNUM** pseudocolumn and the primary key column (**Date**) of the **Period** table (if the **Period** table was segmented, you would also need to find the segment name that is stored in the **RBW_SEGNAME** pseudocolumn). The following query returns the range boundaries for this STAR index:

```
select rbw_rownum, date
   from period
   where date = '04-01-99'
      or date = '07-01-99'
      or date = '10-01-99'
      or date = '01-01-00'
      or date = '04-01-00'
      or date = '07-01-00'
      or date = '10-01-00'
      or date = '01-01-01'
      or date = '04-01-01';
```

The constraints on the **Date** column in the WHERE clause of this query correspond to the range boundaries from the CREATE TABLE statement for the **Sales** table.

The following figure illustrates how sales data is mapped to the data segments and to the STAR index segments.

Figure D-6
Sales Data Mapped to Data and STAR Index Segments



Each segment includes the lower end of its range but excludes the upper end. For example, the segment **s_1q01** includes the data for 2001-01-01 but does not include data for 2001-04-01. Similarly, the STAR index segment **star_1q01** includes the index entries for 2001-01-01 data but does not include index entries for 2001-04-01 data.

The last segment, **s_max**, is designated to contain data with a date of April 1, 2001 or later. At this time, however, the segment is empty because no such data has yet been loaded.

The Data

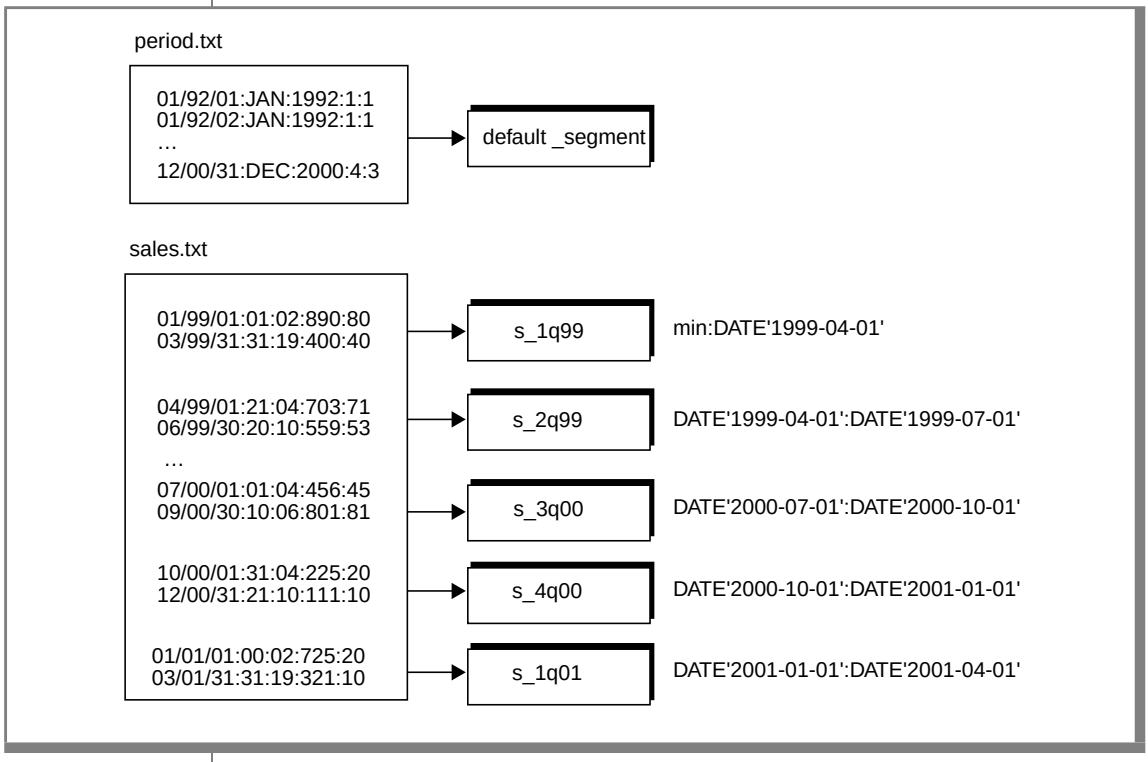
Assume data is loaded into the **Sales** and **Period** tables as follows:

```
load data
   inputfile 'period.txt'
   replace
   separated by ':'
   discardfile 'period_discards'
   discards 5
   into table period (
   perkey date 'MM/Y*/DD',
   month char,
   year integer external,
   quarter integer external,
   tri integer external
);
load data
   inputfile 'sales.txt'
   replace
   separated by ':'
   discardfile 'sales_discards'
   discards 5
   into table sales (
   perkey date 'MM/Y*/DD',
   prodkey integer external,
   mktkey integer external,
   dollars integer external
);
```

The **Product** and **Market** tables must be loaded before the **Sales** table. However, the contents of those tables are not relevant to this example.

The following figure illustrates the data format in the **Period** and **Sales** tables and how the data is distributed among the segments. The index for the **Period** table resides in a separate default segment, and the STAR index for the **Sales** table is segmented like the data, as shown in Figure D-6 on page D-12.

Figure D-7
Data Format for Period and Sales Tables



Rolling Off and Reusing Data and Index Segments

Assume that the database holds a rolling nine quarters of data at a time. When it is time to add the data for the second quarter of 2001, you can remove the data from the first quarter of 1999. This section shows a procedure to reuse the **s_1q99** data and **star_1q99** index segments for the data and index entries for the second quarter of 2001.

This procedure only works if you have the data and the indexes segmented identically, as the previous section outlines.

To reuse data and index segments for data for another year

1. Take the segment containing the oldest data offline.

   ```
   alter segment s_1q99 of table sales offline;
   ```

2. Detach the segment from the table. (This action removes all data from the segment, so make sure you want to do this.)

   ```
   alter segment s_1q99 of table sales detach override
   fullindexcheck on segments (star_1q99):
   ```

   The OVERRIDE FULLINDEXCHECK clause is designed to take advantage of the fact that all index entries corresponding to the **s_1q99** data segment are stored in the **star_1q99** index segment. This arrangement dramatically speeds the performance of the DETACH operation. If your data and indexes are not identically segmented, do not include the OVERRIDE FULLINDEXCHECK clause in your DETACH operation. Omitting the clause ensures that any index entries in other parts of the index are removed.

3. Take the index segment corresponding to the oldest data offline.

   ```
   alter segment star_1q99 of index sales_star offline;
   ```

4. Detach the index segment from the index. (This action removes all index entries from the segment.)

   ```
   alter segment star_1q99 of index sales_star detach;
   ```

5. Rename the old data segment, which now holds data for the second quarter of 2001.

```
alter segment s_1q99 rename s_2q01;
```

6. Rename the old index segment.

```
alter segment star_1q99 rename star_2q01;
```

7. Make any other needed changes to the segments. For example, change the maximum size or path of a PSU, or add a new PSU to the segment.

8. Attach the newly renamed data segment to the table.

```
alter segment s_2q01 attach to table sales
   range (DATE'2001-04-01':DATE'2001-07-01');
```

   The range of the **s_max** segment automatically moves to the range.

```
DATE'2001-07-01':max
```

   Attaching a segment automatically sets the segment to ONLINE mode.

   The **Period** table must contain rows corresponding to the days in the new quarter. Otherwise, new data inserted in the **Sales** table would be discarded due to referential integrity failure.

9. Attach the newly renamed index segment to the table.

```
alter segment star_2q01 attach to index sales_star
   range (821:899);
```

10. You can now use the TMU or SQL INSERT statements to populate the **Sales** table with data for the second quarter of 2001.

Adding a New Segment

As the end of Q2 2001 approaches, you want to add another segment between **s_2q01** and **s_max** to hold data for the next quarter. Because you do not have any unused segments available, create a segment and attach it to the **Sales** table. As data is entered for Q3 2001, it is placed in this new segment.

If you had an extra segment available, for example, after deleting the oldest quarter, you could reuse that segment and avoid the work of creating a new one:

1. Create a new segment named **s_3q01**.

UNIX

```
create segment s_3q01
   storage '/disk10/s1' maxsize 200,
   storage '/disk10/s2' maxsize 200,
   storage '/disk10/s3' maxsize 200;
```

♦

WIN NT/2000

```
create segment s_3q01
   storage 'c:\disk10\s1' maxsize 200,
   storage 'c:\disk10\s2' maxsize 200,
   storage 'c:\disk10\s3' maxsize 200;
```

♦

2. Add the additional **Perkey** entries to the **Period** table to cover the additional date ranges for quarter 3 of 2001. Omitting this step causes referential integrity failures when the data for the new quarter is loaded. For example:

```
insert into period values
   (DATE'2001-07-01', 'JULY', 2001, 3, 2);
...
insert into period values
   (DATE'2001-08-01', 'AUG', 2001, 3, 2);
...
insert into period values
   (DATE'2001-09-01', 'SEPT', 2001, 3, 3);
...
```

3. Attach the **s_3q01** segment to the **Sales** table, specifying the date range for quarter 3 of 2001.

```
alter segment s_2q01 attach to table sales
   range (DATE'2001-07-01':DATE'2001-10-01');
The range of the s_max segment automatically moves to the
range:
DATE'2001-10-01':max
```

   Attaching a segment automatically sets the segment to ONLINE mode.

4. Create a new index segment named **star_3q01**.

UNIX

```
create segment star_3q01
   storage '/disk10/star1' maxsize 200,
   storage '/disk10/star2' maxsize 200,
   storage '/disk10/star3' maxsize 200;
♦
```

WIN NT/2000

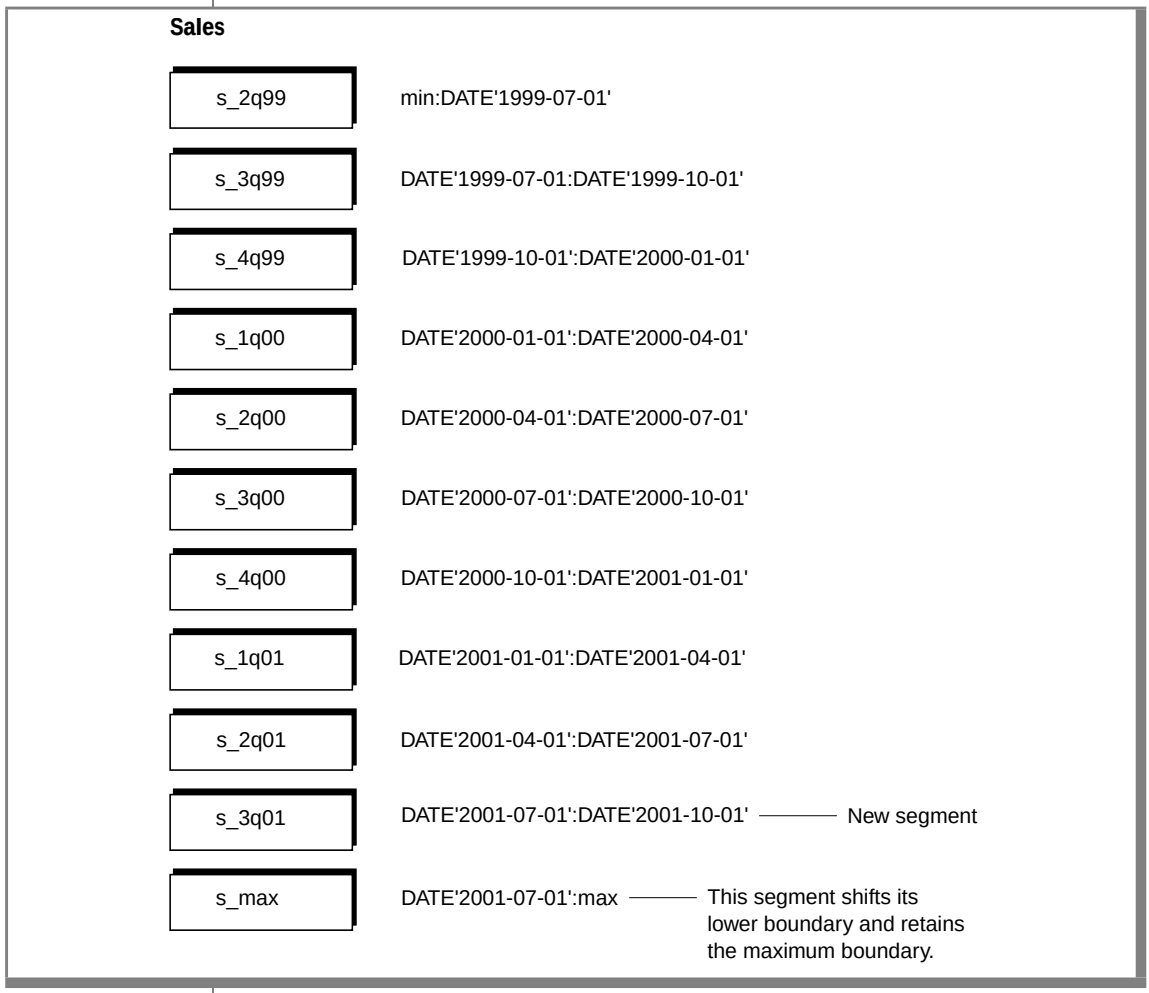
```
create segment star_3q01
   storage 'c:\disk10\star1' maxsize 200,
   storage 'c:\disk10\star2' maxsize 200,
   storage 'c:\disk10\star3' maxsize 200;
♦
```

5. Attach the new index segment to the index.

```
alter segment star_3q01 attach to index sales_star
   range (899:992);
```

The following figure illustrates the new segment ranges for the **Sales** table.

Figure D-8
New Segment Ranges for Sales Table



Now you can load data into the new segment, using either the standard or the offline load procedure.

Using an Offline Load Operation

Assume you want to perform an offline load operation to load a batch of data in the new segment **s_3q01**.

1. Set the segment to OFFLINE mode.

   ```
   alter segment s_3q01 of table sales offline;
   ```

2. Create an extra segment to provide working space for building indexes.

   ```
   create segment work01 storage 'work01' maxsize 100;
   ```

3. Create a TMU control file (**s3q01_input**) that contains:

   - A LOAD DATA statement to read the data input file (**sales_01_data**) and map each field in a column in the offline segment
   - A SYNCH statement to synchronize the indexes of the **Sales** table with the new data

   ```
   load data
      inputfile 'sales_01_data'
      append
      separated by ':'
      discardfile 'discards_sales_01'
      discards 3
      into offline segment s_3q01 of table sales
      working_space work01 (
         perkey date 'MM/Y*/DD',
      prodkey integer external,
      mktkey integer external,
      dollars integer external
      );
   synch offline segment s_3q01 with table sales
      discardfile 'discards_synch';
   ```

   Because the SYNCH operation locks the table, you might prefer to load the data with one control statement and perform the SYNCH operation with another control statement at another time.

4. Log in as the **redbrick** user.
5. Verify that the directory ***redbrick_dir*/bin** on UNIX or ***redbrick_dir*\bin** on Windows is in your path.
6. Change to the directory that contains the **s3q01_input** file.

7. Invoke the TMU and load the new data.

```
rb_tmu -d database_name s_3q01_input system password
```

   The segment **s_3q01** now contains the new data.

8. Set the segment **s_3q01** to ONLINE mode.

```
alter segment s_3q01 of table sales online;
```

9. Drop the working segment.

```
drop segment work01;
```

Deleting the Oldest Data

Now you need to remove data for the oldest quarter, the second quarter of 1999, which resides in the **s_2q99** segment.

To remove the data

1. If you want to save the data, use the TMU to unload the segment to a file or tape.
2. Invoke RISQL (or the tool you use for administrative activities), connecting the database that you want to modify.
3. Set the **s_2q99** segment to OFFLINE mode.

```
alter segment s_2q99 of table sales offline;
```

4. Use one of these statements to detach the **s_2q99** segment.

```
alter segment s_2q99 of table sales detach;

alter segment s_2q99 of table sales detach
   override fullindexcheck on segments (star_2q99);
```

   For a caution about OVERRIDE FULLINDEXCHECK, see "Rolling Off and Reusing Data and Index Segments" on page D-15.

   The data that resided in the **s_2q99** segment is now deleted from the database. The segment is detached from the table but still exists (empty) for reuse. The boundaries of the next segment, **s_3q99**, automatically change to cover the minimum boundary with a range.

```
min:DATE'1999-10-01'
```

5. Modify the **Period** table by deleting the period keys for the old data (second quarter of 1999).

```
delete from period where perkey < DATE'1999-07-01';
```

6. Detach the **star_2q99** index segment.

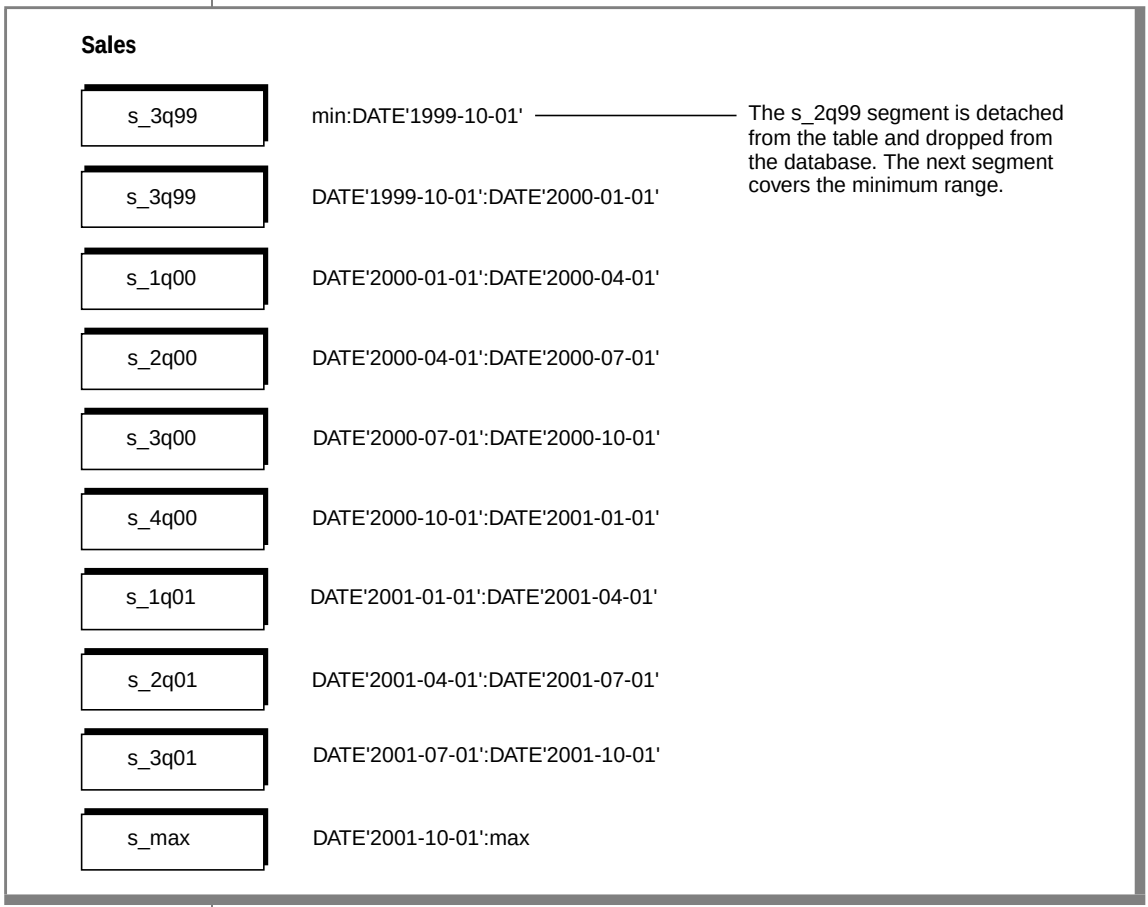
```
alter segment star_2q99 of index sales_star detach;
```

7. You can either save these segments for reuse or drop them with DROP statements.

```
drop segment s_2q99;
drop segment star2q99;
```

The ranges are now as follows.

Figure D-9
Segments and Data Ranges for Sales Table



Reusing the Segments

If you chose not to drop the old segment **s_2q99**, you can reuse it when you need a new segment for Q4 2001 data.

To reuse the segment

1. If you want to rename this segment, use the ALTER SEGMENT statement.

   ```
   alter segment s_2q99 rename s_4q01;
   ```

2. Attach it to the **Sales** table with the new range.

   ```
   alter segment s_4q01 attach to table sales
       range (DATE'2001-10-01':DATE'1998-01-01');
   ```

   The **Period** table must contain rows corresponding to the days in the new quarter. Otherwise, new data inserted in the **Sales** table is discarded due to referential integrity failure.

3. Make any other needed changes; for example, changing MAXSIZE or PATH or adding new storage.

4. Rename the old index segment.

   ```
   alter segment star_2q99 rename star_4q01;
   ```

5. Make any other needed changes to the segments; for example, changing MAXSIZE or PATH or adding new storage.

6. Attach the newly renamed index segment to the table.

   ```
   alter segment star_4q01 attach to index sales_star
      range (992:1084);
   ```

7. The data and index segments are now ready to load data using either the TMU or INSERT statements.

Index

A

Access authorizations 2-37, 7-3 to 7-37
ACCESS_ADVISOR_INFO task
  authorization 7-12
ACCESS_ANY task
  authorization 7-12
ACCESS_SYSINFO task
  authorization 7-12, 8-4
Accounting
  changing modes 8-41
  configuring 8-38
  files 8-37
  job mode 8-34
  LA parameters 8-35
  overview 8-34
  process 8-35
  rbwlogd daemon 8-35
  record format 8-35
  sample code for 8-37
  setting mode 8-40
  starting 8-41
  startup state 8-38
  stopping 8-41
  switching files 8-41
  workload mode 8-34
Accounting file
  location 8-38, 8-39
  removing 8-38
  size 8-37, 8-39
  switching 8-41
ACCOUNTING parameter
  rbw.config file entry B-23
  usage 8-38
Accounts. *See* User accounts.
ACCT_DIRECTORY parameter
  rbw.config file entry B-23
  usage 8-38, 8-39
ACCT_LEVEL parameter
  rbw.config file entry B-24
  usage 8-40
ACCT_MAXSIZE parameter
  rbw.config file entry B-23
  usage 8-37, 8-39
ADD STORAGE clause, ALTER SEGMENT statement 9-29
ADMIN ADVISOR_LOGGING parameter
  modifying 9-53, 9-56
  rbw.config file entry B-25
  usage 8-31
ADMIN database
  described 8-4
  rbw.config file entry B-27
ADMIN parameters
  entries in rbw.config B-23
  modifying 9-55, 9-58
  summary of B-33
Administration daemon
  output 2-37
  See also rbwadmd daemon.
Administration database 8-4
Administration of databases, overview 1-14
Administrative account. *See* redbrick user account.
Administrator tool 1-8, 2-25

Advisor
described 2-14
log files
specifying directory for 8-32
specifying size of 8-33
log files, removing 8-30
logging 8-29
Advisor operator 10-48
ADVISOR_LOGGING, ADMIN parameter
rbw.config file entry B-25
ADVISOR_LOGGING, OPTION parameter
rbw.config file entry B-13
ADVISOR_LOG_DIRECTORY parameter
rbw.config file entry B-25
usage 8-32
use with log files 8-30
ADVISOR_LOG_MAXSIZE parameter
rbw.config file entry B-25
usage 8-33
use with log files 8-29
ADVISOR_SAMPLE_SIZE parameter
modifying 9-55, 9-58
rbw.config file entry B-21
Aggregate maintenance 2-14
Aggregate tables
described 2-14
example 3-29
ALLOW_POSSIBLE_DEADLOCK parameter
modifying 9-54, 9-57
rbw.config file entry B-12
syntax and usage 9-12
ALTER DATABASE statement
CLEAN VERSION LOG clause 6-19
FREEZE QUERY REVISION clause 6-14
START CLEANING clause 6-26
STOP CLEANING clause 6-25
UNFREEZE QUERY REVISION clause 6-22
ALTER SEGMENT statement
ADD STORAGE clause 9-29
ATTACH clause 9-29
CHANGE EXTENDSIZE clause 9-30
CHANGE MAXSIZE clause 9-30
CHANGE PATH clause 9-30, 9-34
CLEAR clause 9-29
DETACH clause 9-29
failure 9-30
FORCE INTACT clause 9-30, 9-36
MIGRATE TO clause 9-31
OFFLINE clause 9-29
ONLINE clause 9-37
OVERRIDE FULLINDEXCHECK clause D-15
RANGE clause 9-29
SEGMENT BY clause 9-30
usage 9-29
VERIFY clause
cause of damage 9-37
damaged segments 9-35
description 9-30
determining segment damage 9-34
ALTER SYSTEM statement
activating a database 8-10
Advisor logging 8-32
changing modes 8-41
clauses
ADVISOR_LOGGING 8-32
CANCEL USER COMMAND 8-11
CHANGE ACCOUNTING LEVEL 8-41
CHANGE LOGGING LEVEL 8-28
CHANGE USER PRIORITY 8-12
CLOSE USER SESSION 8-11
QUIESCE DATABASE 8-10
RESET STATISTICS 8-11
RESUME DATABASE 8-10
START ACCOUNTING 8-41
START LOGGING 8-28
STOP ACCOUNTING 8-41
STOP LOGGING 8-28
SWITCH ACCOUNTING FILE 8-41
SWITCH LOGGING FILE 8-28
SWITCH_ADVISOR_LOG_FILE 8-32
TERMINATE LOGGING DAEMON 8-28
closing a session 8-11
required authorizations 8-9
ALTER TABLE statement
adding column 9-41
adding foreign key 9-44
cascade deletes 9-43
changing default value 9-42
DROP CONSTRAINT clause 9-44
dropping column 9-41
fill factor 9-45
interrupted operations 9-46
ON DELETE clause 9-43
out of space errors 9-47
renaming column 9-42
start and step values for SERIAL columns 9-45
usage 9-40
ALTER USER statement 8-12
ALTER_ANY task
authorization 7-12
ALTER_OWN task
authorization 7-13
ALTER_SYSTEM task
authorization 7-12, 8-4
ALTER_TABLE_INTO_ANY task
authorization 7-13
ARITHABORT parameter
modifying 9-54, 9-57
rbw.config file entry B-11
Arithmetic errors, options B-11
Aroma database
building, tutorial A-1 to A-29
installation 1-20
rbw.config file entry B-27
ASCII code set 2-29
Associative tables 3-12
ATTACH clause, ALTER SEGMENT statement 9-29
Attaching segments 9-31
Attributes, influence on schema design 3-25
Audit events 8-22

Auditing. *See* Event logging.
Authorizations. *See* Task authorizations.
Automatic Row Generation, TMU and referential integrity 2-40
AUTOROWGEN parameter
  modifying 9-55, 9-58
  rbw.config file entry B-11
Availability of tables and indexes 9-32

B

Backup operations
  cleaning version log 6-19
  during index creation 5-16
  on versioned databases 6-24
  overview 1-19
Binary character comparisons 2-29
Bit vector sort operator 10-49
Blocks of data, in version log 6-11
Block, 8-kilobyte 9-20
Browsing dimension tables 3-24
B-TREE
  1-1 match operator 10-48
  scan operator 10-48
  See also Indexes, B-TREE.
Buffer cache size 10-18

C

Cache I/O statistics 8-7
CASCADE keyword, usage 5-15
Cascaded delete operations 2-41, 5-15
Cases, tracked by customer support 12
CATEGORY value, macro 5-22
Cautions
  FORCE INTACT for segments 9-35
  OVERRIDE FULLINDEXCHECK D-15
  OVERRIDE REFCHECK 5-16
CHANGE ACCOUNTING LEVEL clause, ALTER SYSTEM statement 8-41
CHANGE EXTENDSIZE clause, ALTER SEGMENT statement 9-30
CHANGE MAXSIZE clause, ALTER SEGMENT statement 9-30
CHANGE PATH clause, ALTER SEGMENT statement 9-30, 9-34
CHANGE_MINIMUM_DAYS parameter B-26
Character data 2-29
Character encoding. *See* Code set.
CHECK INDEX
  file permissions B-12
  obtaining size information 9-14
Check operator 10-49
CHECK TABLE
  checking table structure 9-14 to 9-18
  file permissions B-12
  VARCHAR fill factor 10-35
  VERBOSE option 9-14 to 9-18, 10-35
CHECK_REPORT_FILE_PERMISSIONS parameter
  modifying file 9-53, 9-56
  rbw.config file entry B-12
CHECK_TABLE_INDEX_DIRECTORY parameter
  modifying 9-54, 9-57
  rbw.config file entry B-12
Choose plan operator 10-50
CLEAN VERSION LOG clause, ALTER DATABASE statement 6-19
Cleanup, temporary space 10-17
CLEANUP_SCRIPT parameter
  modifying 9-56
  modifying file 9-53
  rbw.config file entry B-10
  syntax and usage 10-17
CLEAR clause, ALTER SEGMENT statement 9-29
Clearing segments 9-33
Client tools
  locales for 2-33
  RISQL Entry Tool and RISQL Reporter 2-33
  third-party 2-33
Client/server environment
  compatibility 2-35
  different locales in 2-32
Code set
  ASCII 2-29
  conversions 2-33, 2-35
  defined 2-28
  effect of changing 2-32
Collation sequence 2-29
Columns
  adding 9-41
  changing data type 9-43
  default values, changing 9-42
  dropping 9-41
  estimating width 4-21
  renaming 9-42
Comment icons 10
COMMENT value, macro 5-22
Complete family, defined 2-41
COMPLEX_NUM_ALPHA parameter
  rbw.config file entry B-26
  usage 7-34
COMPLEX_NUM_NUMERICS parameter
  rbw.config file entry B-26
  usage 7-34
COMPLEX_NUM_PUNCTUATION parameter
  rbw.config file entry B-26
  usage 7-35
Configuration file
  described B-1
  entries in B-9
  examples B-2
  modifying 9-52
  parameter summary B-27
Configuration parameter
  ACCOUNTING 8-38, B-23
  ACCT_DIRECTORY 8-38, 8-39, B-23
  ACCT_LEVEL 8-40, B-24
  ACCT_MAXSIZE 8-37, 8-39, B-23
  ADMIN ADVISOR_LOGGING 8-31
  ADVISOR_LOG_DIRECTORY 8-30, 8-32, B-25
  ADVISOR_LOG_MAXSIZE 8-29, 8-33, B-25

ADVISOR_SAMPLE_SIZE B-21
ALLOW_POSSIBLE_
DEADLOCK B-12
ARITHABORT B-11
AUTOROWGEN B-11
CHANGE_MINIMUM_
DAYS B-26
CHECK_REPORT_FILE_
PERMISSIONS B-12
CHECK_TABLE_INDEX_
DIRECTORY B-12
CLEANUP_SCRIPT B-10
COMPLEX_NUM_ALPHA B-26
COMPLEX_NUM_
NUMERICS B-26
COMPLEX_NUM_
PUNCTUATION B-26
COUNT_RESULT B-12
CROSS_JOIN B-12
DEFAULT_DATA_
SEGMENT B-22
DEFAULT_INDEX_
SEGMENT B-22
DEFAULT_PSU_
EXTENDSIZE B-23
DEFAULT_SEGMENT_
SIZE B-23
EXPIRATION_DAYS B-26
EXPIRATION_WARNING_
DAYS B-26
EXPORT_DEFAULT_PATH B-15
EXPORT_DELIMITER B-15
EXPORT_MAX_FILE_SIZE B-15
FILE_GROUP B-17
FILLFACTOR B-18
FILLFACTOR_VARCHAR B-18
FIRST_DAYOFWEEK B-11
FORCE_AGGREGATION_
TASKS B-17
FORCE_FETCH_TASKS 11-5,
B-17
FORCE_HASHJOIN_
TASKS B-17
FORCE_JOIN_TASKS 11-5, B-17
FORCE_SCAN_TASKS 11-5,
B-16
GRANT_TEMP_RESOURCE_
TO_ALL B-12
GROUP B-17
IDLE_TIMEOUT B-15
IGNORE_OPTICAL_
INDEXES 9-40, B-22
IGNORE_PARTIAL_
INDEXES B-22
INDEX_TEMPSPACE_
DIRECTORY 4-37, B-18
INDEX_TEMPSPACE_
MAXSPILLSIZE 4-37, B-18
INDEX_TEMPSPACE_
THRESHOLD 4-37, 4-38, B-18
INFO_MESSAGE_LIMIT B-21
INTERVAL B-11
LOCALE B-11
LOCK_FAILED_
ATTEMPTS B-26
LOCK_PERIOD_HOURS B-26
LOGFILE_SIZE B-11
LOGGING 8-25, B-24
LOG_AUDIT_LEVEL 8-27, B-24
LOG_DIRECTORY 8-24, 8-25,
B-24
LOG_ERROR_LEVEL 8-27, B-24
LOG_MAXSIZE 8-24, 8-26, B-24
LOG_OPERATIONAL_
LEVEL 8-27, B-24
LOG_SCHEMA_LEVEL 8-27,
B-24
LOG_USAGE_LEVEL 8-27, B-25
MAPFILE B-10
MAX_ACTIVE_
DATABASES B-11
MAX_SERVERS B-10
MESSAGE_DIR B-11
MINIMUM_LENGTH B-26
NLS_LOCALE LOCALE 2-31
NLS_LOCALE MESSAGE_
DIR 2-36
OPTICAL_AVAILABILITY 9-39,
B-23
OPTION ADVISOR_
LOGGING B-13
PARALLEL_HASHJOIN B-17
PARTIAL_AVAILABILITY B-22
PARTITIONED_PARALLEL_
AGGREGATION B-17
PRECOMPUTED_VIEW_
MAINTENANCE B-15
PRECOMPUTED_VIEW_
MAINTENANCE_ON_
ERROR B-15
PRECOMPUTED_VIEW_
QUERY_ REWRITE B-13
PROCESS_CHECKING_
INTERVAL B-11
QUERYPROCS B-16
QUERY_MEMORY_LIMIT B-18
QUERY_TEMPSPACE_
DIRECTORY B-19
QUERY_TEMPSPACE_
MAXSPILLSIZE B-19
RANDOM_SAMPLE_
MARGIN B-21
RBW_LOADINFO_LIMIT B-21
READER_PRIORITY 9-7, B-12
RENICE_COMMAND 8-12, B-25
REPORT_INTERVAL 8-14, B-25
RESTRICT_PREVIOUS B-26
RESULT_BUFFER B-17
RESULT_BUFFER_FULL_
ACTION B-17
ROWCOUNT B-21
ROWS_PER_FETCH_TASK B-16
ROWS_PER_SCAN_TASK B-16
SERVER B-10
SERVER_NAME B-11
SHMEM B-10
specifying in rbw.config
TEMPORARY_DATA_
SEGMENT B-22
TEMPORARY_INDEX_
SEGMENT B-22
TMU_BUFFERS B-16
TMU_COMMIT_RECORD_
INTERVAL B-14
TMU_COMMIT_TIME_
INTERVAL B-14
TMU_CONVERSION_
TASKS B-19
TMU_INDEX_TASKS B-20
TMU_INPUT_TASKS B-20
TMU_MAX_TASKS B-20
TMU_MMAP_LIMIT B-20
TMU_OPTIMIZE B-18
TMU_ROW_MESSAGES B-19
TMU_SERIAL_MODE B-19
TMU_VERSIONING B-14

TOTALQUERYPROCS B-16
TRANSACTION_ISOLATION_LEVEL B-13
tuning 11-4
UNIFIED_LOGON B-10
UNIFORM_PROBABILITY_FOR_ADVISOR B-13
USE_INVALIDATE_PRECOMPUTED_VIEWS B-13
VERSIONING B-13
See also individual parameter names.
CONNECT system role, described 2-38, 7-7
Connections limit
changing 9-53
rbw.config file entry B-10
CONSTRAINT keyword 5-15
Constraints, weakly selective 4-10
Contact information 17
Control-c coordination thread 1-14
Controlling database activity 8-9
Conventions
syntax diagrams 7
syntax notation 6
Copy management utility 8-4, 9-48
Correlated subquery 10-85
Costs, of version log 6-7
COUNT function B-12
COUNT_RESULT parameter
modifying 9-54, 9-57
rbw.config file entry B-12
CPUs, allocation for parallel queries 11-37
CREATE INDEX statement, usage 5-16
CREATE MACRO statement, usage 5-21
CREATE ROLE statement, usage 7-15
CREATE SEGMENT statement, usage 5-12
CREATE TABLE statement
constraint names 5-15
examples A-8
usage 5-13
CREATE VIEW statement, usage 5-19
CREATE_ANY task
authorization 7-12
CREATE_OWN task
authorization 7-13
Creating a database, Aroma tutorial A-1 to A-29
Cross-reference tables 3-12
CROSS_JOIN parameter
modifying 9-54, 9-57
rbw.config file entry B-12
CURRENT_USER, SQL variable 5-21
Custom roles. *See* User-created roles.

D

Daemon processes. *See* Processes.
Data
aggregation 3-29
loading 2-3
organizing 4-3
referential integrity during delete 2-41, 5-15
Data referential integrity during insert and load 2-40
Data source name definitions 2-24
Data types
changing column 9-43
sizes of C-39
Database administrator
DBA account 2-37
role 1-19
Database directory, defined 2-15
Database server
described 1-6
installation 2-21
limits 1-20
maximum number of B-10
monitoring on UNIX 9-59
monitoring on Windows 9-62
on Windows 1-6
software components 1-5
Databases
access to 7-3 to 7-37
activating 8-10
adding new 5-8
adding users 7-5
backup and restore with versioning 6-24
changing passwords 7-6, A-8
controlling activity 8-9
copying 9-34, 9-48
creating 1-16
example, Aroma A-1 to A-29
procedure 5-3 to 5-25
default segment location 5-12
deleting 1-16, 9-68
design 1-15
dropping database objects 9-64, 9-68
freezing 6-21
growth patterns 4-48
implementing 1-16, 2-5
limits and maximum sizes 1-20
loading data A-24
locking 9-6 to 9-13
logical names 2-16
defining 5-8
rbw.config file entry B-27
maintaining 1-19
monitoring activity 8-6
moving 9-34, 9-48
organizing data into tables 4-3
password security 7-26 to 7-37
quiesce 8-10
sizing example 4-27
system account and password 5-9
table and index sizes, estimating 4-20
tuning 1-19
unfreezing 6-22
when to lock 9-10
DBA. *See* Database administrator.
DBA system role
described 2-38, 7-7
granting 7-8, 7-18
revoking 7-8, 7-21
dbcreate utility 5-4
deleting databases 9-68
example A-7

dbsize utility 4-21
DDL operations 9-7
Deadlocks 9-12
Decision-support databases 3-5
DEFAULT parameters
  entries in rbw.config B-21
  summary of B-33
DEFAULT_DATA_SEGMENT parameter
  modifying 9-54, 9-57
  rbw.config file entry B-22
  syntax and usage 10-22
  with multiple databases 5-12
DEFAULT_INDEX_SEGMENT parameter
  modifying 9-54, 9-57
  rbw.config file entry B-22
  syntax and usage 10-22
  with multiple databases 5-12
DEFAULT_PSU_EXTENDSIZE parameter
  described 2-10
  modifying 9-53, 9-56
  rbw.config file entries B-23
DEFAULT_SEGMENT_SIZE parameter
  described 2-9
  modifying 9-53, 9-56
  rbw.config file entries B-23
Delete cascade operator 10-51
DELETE operations
  locking tables 9-10
  modes, actions 5-15
  versioned 6-13
Delete operator 10-50
Delete refcheck operator 10-51
Demonstration database, script to install 4
Dependencies, software 4
Design of databases 1-15
DETACH clause, ALTER SEGMENT statement 9-29
Detaching segments 9-31
Dimension table
  cleaning with versioning 6-7
  defined 3-8
  large 4-16
  number of rows in 11-19
Directories for temporary space 4-35, 4-44
Disk groups
  defining 11-6
  effect on parallel queries 11-32
  specifying processes 11-8
Disk space
  allocation 4-48, 5-13
  offline load requirements 4-40
  organizing with segmented storage 4-45
  reuse 9-22
  table and index sizes, estimating 4-20
  table growth, planning for 4-48
  temporary space requirements, estimating 4-37 to 4-44
Disk spill files, removing 10-17
Disk usage, parallel queries 11-34
Divide-by-zero errors, options B-11
Documentation
  list for Red Brick Decision Server 14
  online manuals 16
  printed manuals 16
Domains, for TARGET indexes
  defined 4-10
  specifying size 4-10
  use with TARGETjoin 4-18
DROP HIERARCHY statement, usage 9-66
DROP ROLE statement, usage 9-66
DROP statement 9-65
DROP_ANY task
  authorization 7-12
DROP_OWN task
  authorization 7-13
DST_COMMANDS table C-26
DST_DATABASES table
  column names C-28
  version log space 6-23
DST_LOCKS table 9-8, C-31
DST_SESSIONS table C-32
DST_USERS table C-35
Dynamic statistic tables
  and RBW_TABLES table 8-6
  DST_COMMANDS C-26
  DST_DATABASES C-28
  DST_LOCKS C-31
  DST_SESSIONS C-32
  DST_USERS C-35
  overview 8-6
  refresh interval 8-14
  statistic collection interval 8-14

E

Environment variables
  boldface type 5
  ODBCINI 2-24
  RB_CONFIG 2-25
  RB_DSN 2-25
  RB_EXE 2-25
  RB_HOME 2-25
  RB_HOST 2-25
  RB_NLS_LOCALE 2-33
  RB_PATH 2-25
  RISQL Entry Tool 7-4
  user accounts 7-4
Error events 8-22
Event logging
  categories 8-21, 8-22
  changing severity 8-28
  configuring 8-25
  log daemon 8-16
  log files 8-24
  logging subsystem 8-16
  message templates 8-17
  messages 8-21
  overview 8-15
  severity levels 8-21
  starting 8-28
  startup state 8-25
  USAGE ROUTINE 8-23
Exchange operator 10-52
Execute operator 10-52
EXPAND statement 5-24
Expiration, passwords 7-28
EXPIRATION_DAYS parameter
  rbw.config file entry B-26
  usage 7-28, 7-32
EXPIRATION_WARNING_DAYS parameter
  rbw.config file entry B-26
  usage 7-30
Expired user accounts 7-29

EXPLAIN statement 10-57
EXPORT task authorization 7-12
EXPORT_DEFAULT_PATH parameter
  modifying 9-54, 9-57
  rbw.config file entry B-15
EXPORT_DELIMITER parameter
  modifying 9-54, 9-57
  rbw.config file entry B-15
EXPORT_MAX_FILE_SIZE parameter
  modifying 9-54, 9-57
  rbw.config file entry B-15
Extended star schema 3-22
EXTENDSIZE value
  RBW_TABLES table 9-33

F

Fact table 3-8, 3-25
Fact table selectivity, estimating 10-77
Fact-to-fact joins
  rules for STARjoin 4-5
  using synonyms with 10-81
Feature icons 10
File-system-full messages 9-18
FILE_GROUP parameter
  described 11-4
  modifying 9-54, 9-57
  rbw.config file entry B-17
  syntax and usage 11-6
Fill factors, index
  changing 10-41
  described 4-23
  effect on key size 10-42
  setting 10-38
Fill factor, VARCHAR
  setting 10-29
Fill factor, varchar
  changing 9-45
FILLFACTOR parameters
  modifying 9-55, 9-58, 10-37, 10-43
  rbw.config file entries B-18
  usage 10-40
FILLFACTOR_VARCHAR parameter, rbw.config file entry B-18
Findserver utility
  INTERVAL B-11
  usage 9-61
FIRST_DAYOFWEEK parameter
  modifying 9-54, 9-57
  rbw.config file entry B-11
FOR DELETE option 9-10
FORCE INTACT clause, ALTER SEGMENT statement 9-30, 9-35, 9-36
FORCE SCAN TASKS parameter
  usage 11-26
FORCE_AGGREGATION_TASKS parameter
  described 11-5
  modifying 9-54, 9-57
  rbw.config file entry B-17
FORCE_FETCH_TASKS parameter
  described 11-4
  modifying 9-54, 9-57
  rbw.config file entry B-17
  usage 11-27
FORCE_HASHJOIN_TASKS parameter
  described 11-29
  modifying 9-54, 9-57
  rbw.config file entry B-17
FORCE_JOIN_TASKS parameter
  described 11-4
  modifying 9-54, 9-57
  rbw.config file entry B-17
  usage 11-27
FORCE_SCAN_TASKS parameter
  described 11-4, 11-5
  modifying 9-54, 9-57
  number of parallel tasks 11-26
  rbw.config file entry B-16
FOREIGN KEY REFERENCES clause, relating tables 4-49
Foreign keys
  adding and dropping 9-44
  defined 3-8
  multicolumn 3-12, 4-17
FREEZE QUERY REVISION clause, ALTER DATABASE statement 6-14
Frozen versions
  controlling 6-21
  defined 6-13
  operations allowed in 6-15
  operations not allowed in 6-15
  overriding 6-22
Functional join operator 10-52

G

General purpose operator 10-53
GRANT authorization and role command
  examples 7-19
  usage 7-8, 7-18
GRANT CONNECT statement 7-5, 7-6
GRANT privilege statement
  examples 7-17
  usage 7-9
GRANT_OWN task authorization 7-13
GRANT_TABLE task authorization 7-12
GRANT_TEMP_RESOURCES_TO_ALL parameter
  modifying 9-56
GRANT_TEMP_RESOURCE_TO_ALL parameter
  modifying 9-53
  rbw.config file entry B-12
GROUP parameter
  described 11-4
  modifying 9-54, 9-57
  rbw.config file entry B-17
  syntax and usage 11-8
GUI tool, Administrator 2-25

H

Hash
  1-1 match operator 10-53
  AVL aggregate operator 10-53
  join 4-12
Hierarchies, dropping 9-66
High recoverability versioning
  described 6-16
  with frozen versions 6-22

I

Icons
  feature 10
  important 10
  platform 10
  tip 10
  warning 10
IDLE_TIMEOUT parameter
  modifying 9-53, 9-56
  rbw.config file entry B-15
IGNORE_OPTICAL_INDEXES parameter
  modifying 9-54, 9-57
  rbw.config file entry B-22
  syntax and usage 9-40
IGNORE_PARTIAL_INDEXES parameter
  modifying 9-54, 9-57
  rbw.config file entry B-22
  syntax and usage 10-28
IGNORE_QUIESCE task authorization 7-12
Immediate family, defined 2-41
Important paragraphs, icon for 10
Incremental loading 6-5
Indexes
  B-TREE
    attaching segments 9-31
    defined 2-6
    example 4-9
    size estimates 4-26
  creation guidelines 5-16
  dropping 9-65
  estimating sizes 4-22
  growth, monitoring 9-18
  loading tables with 5-18
  overview 2-6
  parallel 5-17
  partial availability 2-13, 10-28
  segmenting column 9-32
  selection 10-28
  selection with optical segments 9-40
  STAR
    attaching segments 9-31
    creating 5-19
    defined 2-6
    effect of adding rows 4-49
    examples 4-6
    fill factor, changing 10-43
    invalid, cause 4-49
    multiple 2-6
    performance with 10-43
    selectivity estimates for preloaded dimensions 10-79
    size estimates 4-26
    size without MAXROWS PER SEGMENT 4-49
    use of MAXROWS PER SEGMENT option 5-14
    use of MAXSEGMENTS option 5-14
    with simple star schema 3-8
  TARGET
    attaching segments 9-31
    choosing domain size 4-10
    defined 2-6
    domains, defined 4-10
    example 4-10
    hybrid 4-11, 4-18
    overview 5-19
    performance 5-19
    selectivity 4-10
    size estimates 4-26
    temporary space requirements 4-42
    use with TARGETjoin 4-12
  verifying creation 5-18
  when to create 4-4
INDEX_TEMPSPACE parameters
  current values 10-16
  described 4-36
  determining values for 4-36
  DIRECTORY 4-37
    location 5-17
    rbw.config file entries B-18
    usage 10-12
    use of 10-9
  MAXSPILLSIZE 4-37
    determining size 4-39
    for offline load 4-41
    rbw.config file entries B-18
    usage 10-12
  modifying 9-55, 9-58
  resetting 10-14
  setting 10-7
  THRESHOLD 4-37
    determining value 4-38
    rbw.config file entries B-18
    syntax 10-13
    usage 10-12
    value 10-13
  use in offline load operations 4-40
Indirect role membership
  defined 7-11
  example 7-20, 7-24
  usage 7-18
Informix customer support 11
INFO_MESSAGE_LIMIT parameter
  modifying 9-54, 9-57
  rbw.config file entry B-21
Initialization files 1-17, 2-22
INITSIZE value, RBW_TABLES table 9-33
Insert operator 10-54
Installation, defining locale during 2-31
Installing database server 1-15, 2-21
INTACT column
  values 9-36
Interdimensional OR queries 10-84
INTERVAL parameter
  modifying 9-53
  rbw.config file entry B-11
  usage 9-61
Invalid STAR indexes 4-49
Isolation level 9-12
I/O contention, parallel queries 11-34

J

Java database connectivity 1-8
JDBC API 1-8
JDBC Driver 1-8
Job accounting 8-34
Joins
  algorithms 10-44
  between fact tables, indexes for 4-5
  foreign key references 2-6
  relating tables 4-49

K

Keywords in syntax diagrams 9

L

LA accounting parameters 8-35
Language
  defined 2-28
  effect of changing 2-32, 2-36
  nontranslated text 2-32
Lexical character comparisons 2-30
Limits, database server 1-20
Linguistic character
    comparisons 2-30
Listener thread 1-13
LOAD DATA statement examples,
    Aroma database A-17 to A-25
Load window 6-4
Loading data
  examples, Aroma
      database A-17 to A-28
  incrementally 6-5
  into versioned databases 6-8
  overview 1-18, 2-3
  time-cyclic data,
      example D-1 to D-23
  while database is online 9-7
  with high recoverability 6-16
  with indexes 5-18
  with periodic commit 6-5
LOCALE parameter
  modifying 9-53, 9-56
  rbw.config file entry B-11
Locales
  client tool 2-33
  client/server environment 2-32
  code set component 2-28
  components of 2-27
  default 2-31, 2-34
  defining during installation 2-31
  language component 2-28
  overriding server 2-32
  sort component 2-29
  specifying different for
      databases 5-5
  system table references 2-31
  territory component 2-28
Locked user accounts 7-36
Locking
  READ option 9-6
  types of table locks 9-8
  wait behavior, changing 9-11
  when to lock tables and
      databases 9-6, 9-13
  WRITE option 9-6
Lock-out period, specifying 7-37
LOCK_DATABASE task
    authorization 7-12
LOCK_FAILED_ATTEMPTS
    parameter B-26
  usage 7-36
LOCK_PERIOD_HOURS
    parameter
  rbw.config file entry B-26
  usage 7-37
Log daemon. *See* rbwlogd daemon.
Log files
  Advisor 8-30
  naming conventions 8-24, 8-30
  removing 8-24
  specifying location 8-25
  specifying size 8-26
  switching 8-28
  types on UNIX 9-62
  types on Windows 9-63
Log viewer
  syntax 8-18
  usage 8-19
LOG _MAXSIZE configuration
    parameter 8-24
logdview executable 8-17
LOGFILE_SIZE parameter
  modifying 9-53, 9-56
  rbw.config file entry B-11
Logging. *See* Event logging.
LOGGING configuration
    parameter
  rbw.config file entry B-24
  usage 8-25
Logging queries 8-29
Logical database names
  defining 5-8
  described 2-16
  rbw.config file entries B-27
Logical I/O statistics 8-7
LOG_AUDIT_LEVEL
    configuration parameter
  rbw.config file entry B-24
  usage 8-27
LOG_DIRECTORY configuration
    parameter
  rbw.config file entry B-24
  usage 8-24, 8-25
LOG_ERROR_LEVEL
    configuration parameter
  rbw.config file entry B-24
  usage 8-27
LOG_MAXSIZE configuration
    parameter
  rbw.config file entry B-24
  usage 8-26
LOG_OPERATIONAL_LEVEL
    configuration parameter
  rbw.config file entry B-24
  usage 8-27
LOG_SCHEMA_LEVEL
    configuration parameter
  rbw.config file entry B-24
  usage 8-27
LOG_USAGE_LEVEL
    configuration parameter
  rbw.config file entry B-25
  usage 8-27

M

Macros
  creation and use 5-21
  described 1-17
  dropping 9-66
  examples 5-24
  EXPAND statement 5-24
  resolving references 5-23
  scope 5-23
Maintaining database 1-19
Many-to-many relationship
  defined by fact table 3-12
  defined by multicolumn foreign
      key 3-12
Many-to-one relationship, foreign
    keys 3-9

MAPFILE parameter
modifying 9-53
rbw.config file entry B-10
MAXROWS PER SEGMENT option
and fill factors 10-30
and VARCHAR fill factor 5-14
changing value 9-43
effect on STAR indexes 4-25
error message 5-14
forcing REORG operation 5-14
in TARGET indexes 4-26
rows per block 10-34
unused row numbers 10-34
usage 5-14
value in RBW_TABLES C-19
MAXSEGMENTS option
changing value 9-43
effect on STAR indexes 4-25
forcing REORG operation 5-14
usage 5-14
value in RBW_TABLES C-19
MAXSIZE value
RBW_STORAGE table 9-20
RBW_TABLES table 9-33
MAXSIZE_ROWS value, in RBW_TABLES table C-19
MAXSPILLSIZE value
estimating, for queries 4-44
for INDEX_TEMPSPACE 4-39
for query temporary slices 4-44
syntax 10-14
MAX_ACTIVE_DATABASES parameter
modifying 9-53, 9-56
rbw.config file entry B-11
MAX_SERVERS parameter
modifying 9-53, 9-56
rbw.config file entry B-10
Membership, roles 7-11
Memory use
loading and index requirements 4-37 to 4-44
parallel queries 11-36
Merge phase of optimized index build 4-37
Merge sort operator 10-54
Message templates 8-17
Messages
files 2-36
internal, not translated 2-36
translated 2-36
MESSAGE_DIR parameter
modifying 9-53, 9-56
rbw.config file entry B-11
Microsoft Access, database access with 7-5
MIGRATE TO clause, ALTER SEGMENT statement 9-31
MINIMUM_LENGTH parameter
rbw.config file entry B-26
usage 7-33
MODIFY_ANY task authorization 7-12
Monitoring
database activity 8-6
database server processes 9-59
frozen versions 6-21
growth of tables and indexes 9-18
VARCHAR fill factor 10-35
version log space 6-23
Multibyte character sets 2-29
Multiple databases, default segment location 5-12
Multistar schema 3-14

N

Naive 1-1 match operator 10-54
NLS_LOCALE LOCALE parameter 2-31
NLS_LOCALE MESSAGE_DIR parameter 2-36
NLS_LOCALE parameters
entries in rbw.config B-11
summary of B-28
NO ACTION keyword
described 2-41
usage 5-15
NO WAIT on locks 9-11
NULL values, in ORDER BY clauses 2-23

O

Object privileges
defined 7-9
granting 7-9
granting to roles 7-16
overview 2-38
ODBC
client application requirements 7-5
driver 1-8
initialization files 2-24
setting data sources 2-24
ODBCINI environment variable 2-24
odbc.ini file 2-24
OFFLINE clause, ALTER SEGMENT statement 9-29
Offline load operations
example D-20
space-planning 4-40
OFFLINE_LOAD task authorization 7-13
ONLINE clause, ALTER SEGMENT statement 9-37
Online manuals 16
Online transaction-processing databases (OLTP) 3-4
Operational events 8-23
Operators
advisor 10-48
bit vector sort 10-49
B-TREE 1-1 match 10-48
B-TREE scan 10-48
check 10-49
choose plan 10-50
defined 10-47
delete 10-50
delete cascade 10-51
delete refcheck 10-51
exchange 10-52
execute 10-52
functional join 10-52
general purpose 10-53
hash 1-1 match 10-53
hash AVL aggregate 10-53
insert 10-54
merge sort 10-54
naive 1-1 match 10-54

RISQL calculate 10-54
sample scan 10-54
simple merge 10-55
sort 1-1 match 10-55
STARjoin 10-55
subquery 10-56
table scan 10-56
TARGET scan 10-57
TARGETjoin 10-56
update 10-57
virtual table scan 10-57
Optical segments
defined 9-38
index selection 9-40
Optical storage
support 9-37 to 9-39
OPTICAL_AVAILABILITY parameter
modifying 9-54, 9-57
rbw.config file entry B-23
syntax and usage 9-39
OPTION ADVISOR_LOGGING parameter
modifying 9-54, 9-57
rbw.config file entry B-13
OPTION parameters
entries in rbw.config B-11
summary of B-30
OR versus UNION operation 10-84
Outboard tables 3-13
Out-of-memory errors 4-38, 5-17
Out-of-space errors 4-39
Outrigger tables 3-13
OVERRIDE FULLINDEXCHECK clause, ALTER SEGMENT statement D-15
OVERRIDE REFCHECK clause
caution 2-41
usage 5-16

P

Parallel indexes 5-17
Parallel processing
on demand 11-4
overview 2-4
tuning for specific query types 11-38
Parallel query processing
allocating tasks for 11-27 to 11-29
enabling 11-5
join phase 11-19
limiting processes 11-10
minimum row requirements 11-13
number of PSUs 4-48
reducing disk contention 11-6
STARjoin 11-17
system limitations 11-32
tuning 11-3 to 11-43
PARALLEL_HASHJOIN parameter
rbw.config file entry B-17
Parameter. *See* Configuration parameter.
Partial availability
indexes 10-28
tables 10-25
PARTIAL_AVAILABILITY parameter
modifying 9-54, 9-57
rbw.config file entry 9-32, B-22
syntax and usage 10-26
PARTITIONED_PARALLEL_AGGREGATION parameter
described 11-5
modifying 9-54, 9-57
rbw.config file entry B-17
PASSWORD parameters
CHANGE_MINIMUM_DAYS 7-32
COMPLEX_NUM_ALPHA 7-34
COMPLEX_NUM_NUMERICS 7-34
COMPLEX_NUM_PUNCTUATION 7-35
described 7-27
entries in rbw.config B-26
EXPIRATION_DAYS 7-28
EXPIRATION_WARNING_DAYS 7-30
LOCK_FAILED_ATTEMPTS 7-36
LOCK_PERIOD_HOURS 7-37
MINIMUM_LENGTH 7-33
modifying 9-55, 9-58
RESTRICT_PREVIOUS 7-31
summary of B-34
Password parameters. *See* PASSWORD parameters.
Passwords
changes, enforcing 7-28
changing 7-6
changing system default A-8
complexity and length 7-34
expiration 7-28
expiration warning 7-30
frequency of changes 7-32
limiting reuse 7-31
locked accounts 7-36
lock-out period 7-37
minimum length 7-33
new users 7-5
restrictions, standard 7-6
security 7-26
security parameters 7-27
standard restrictions 5-10
system account, changing 5-9
Pathnames, relative and full 9-48
Performance
general tuning 10-5 to 10-86
parallel query tuning 11-3 to 11-43
parallel table scan 11-41
resource and workload analysis 11-34 to 11-38
schema design 3-8
STARjoin queries 11-38
SuperScan technology 11-42
TARGET indexes 5-19
Periodic commit 6-5
Permission. *See* Object privileges and Databases, access to.
Physical I/O statistics 8-7
Physical storage units. *See* PSUs.
Platform icons 10
Precomputed views 2-14
PRECOMPUTED_VIEW_MAINTENANCE parameter
modifying 9-53, 9-56
rbw.config file entry B-15

PRECOMPUTED_VIEW_
MAINTENANCE_ON_ERROR
parameter
modifying 9-53, 9-56
rbw.config file entry B-15
PRECOMPUTED_VIEW_QUERY_
REWRITE parameter
modifying 9-54, 9-57
rbw.config file entries B-13
Predicate, defined 10-48
Preloaded dimensions and fact
table selectivity 10-77
Primary key, defined 3-8
Printed manuals 16
Private macros 5-23
Process checker daemon 1-12
Processes
allocating, for parallel
queries 11-27 to 11-29
defined 1-9
monitoring 9-59
PROCESS_CHECKING_
INTERVAL parameter
modifying 9-53, 9-56
rbw.config file entry B-11
Pseudocolumns 9-23
PSUs
allocating disk space 5-13
cannot open 9-34
changing location 9-34
changing size 9-33
damaged 9-35
extend size 2-10
for indexes 5-13
for version log 6-11
MAXSIZE value 9-20
moving 9-34
sequence ID C-16
size and growth 2-9
PUBLIC access 7-9
Public macro 5-23
PUBLIC_MACROS task
authorization 7-13

Q

Queries
logging 8-23, 8-29
parallel processes
for 11-5 to 11-29
Query performance
additional indexes 10-43
SQL improvements 10-84
See also Performance and Parallel
query processing.
Query processing 10-44 to 10-60
Query sampling 2-15
Query-priority locking
disabling 9-7
QUERYPROCS parameter
definition 11-4
described 11-5 to 11-27
modifying 9-53, 9-56
rbw.config file entry B-16
syntax and usage 11-10
QUERY_MEMORY_LIMIT
parameter
estimating value of 4-43
modifying 9-54, 9-57
setting 10-18
usage 10-12
value 10-14
QUERY_TEMPSPACE
parameters 10-7 to 10-18
current values 10-16
described 4-43 to 4-44
DIRECTORY
rbw.config file entries B-19
use of 10-9
MAXSPILLSIZE
rbw.config file entries B-19
value 4-44
modifying 9-54, 9-57
QUERY_MEMORY_LIMIT
rbw.config file entries B-18
QUERY_TEMPSPACE_
DIRECTORY parameter
rbw.config file entry B-19
usage 10-12
QUERY_TEMPSPACE_
MAXSPILLSIZE parameter
rbw.config file entry B-19
usage 10-12
Quiesce, database 8-10

R

RAID devices, with version
log 6-18
RANDOM_SAMPLE_MARGIN
parameter
modifying 9-55, 9-58
rbw.config file entry B-21
RANGE clause, ALTER SEGMENT
statement 9-29
RBMON parameter B-27
rbodbc32.ini file 2-24
.rbretrc file 2-23
rbwadmd daemon
details 8-12
output 2-37
starting 9-60
statistic collection interval 8-14
stopping 9-60
rbwadmd thread
starting 9-60
stopping 9-62
RBWAPI parameters
entries in rbw.config B-10
summary of B-28
rbwapid daemon
described 1-11
rbw.config file entries B-10
rbwconc thread 1-14
.rbwerr file 2-23
rbwlogd daemon
defined 1-12
output 2-37
role in accounting 8-35
starting 8-16
stopping 8-28
rbwlogview executable 8-17
rbwlsnr thread 1-13
RBWMON parameter,
INTERVAL B-11

rbwpchk daemon 1-12
.rbwrc file 2-22
rbwsvr process 1-11
rbwvcd daemon. *See* Vacuum cleaner daemon.
rbw.config file. *See* Configuration file.
rbw.findserver utility 9-61
rbw.servermon daemon
  and Findserver 9-61
  interval setting B-11
  stopping 9-61
rbw.show script, usage 9-60
rbw.start script
  rbwadmd daemon 9-60
  usage 9-60
rbw.stop script, usage 9-60
RBW_COLUMNS table C-4
RBW_CONSTRAINTS table 5-15, C-6
RBW_CONSTRAINT_COLUMNS table C-5
RBW_HIERARCHIES table C-6
RBW_INDEXCOLUMNS table C-7
RBW_INDEXES table C-7
RBW_LOADINFO table C-8
RBW_LOADINFO_LIMIT parameter
  modifying 9-54, 9-57
  rbw.config file entry B-21
RBW_MACROS table C-10
RBW_OPTIONS table 2-31, C-11
RBW_PRECOMPVIEW_CANDIDATES table C-2
RBW_PRECOMPVIEW_COLUMNS table C-12
RBW_PRECOMPVIEW_UTILIZATION table C-2
RBW_RELATIONSHIPS table 5-15, C-12
RBW_ROLES table 7-23, C-14
RBW_ROLE_MEMBERS table 7-23, C-13
RBW_ROWNUM pseudocolumn 9-23, 9-31
RBW_SEGID pseudocolumn 9-23
RBW_SEGMENTS table 2-21, C-14
RBW_SEGNAME pseudocolumn 9-23
RBW_STORAGE table C-16
RBW_SYNONYMS table C-17
RBW_TABAUTH table 7-23, C-18
RBW_TABLES table C-19
RBW_USERAUTH table 7-23, C-20
RBW_VIEWS table C-24
RBW_VIEWTEXT table C-25
rb_cm utility 8-4, 9-48
RB_CONFIG environment variable
  described 2-25
  user accounts 7-4
rb_creator utility 5-4, A-7
RB_DEFAULT_LOADINFO file 5-8
rb_deleter utility, deleting databases 9-68
RB_DSN environment variable 2-25
RB_EXE environment variable 2-25
RB_HOME environment variable 2-25
RB_HOST environment variable
  described 2-25
  user accounts 7-4
RB_NLS_LOCALE environment variable 2-33
RB_PATH environment variable
  described 2-25
  user accounts 7-4
rb_sample.cleanup script, usage 10-17
READ locking option 9-6
Read locks 9-8
Read statistics 8-7
READER_PRIORITY parameter
  modifying 9-53, 9-56
  rbw.config file entry B-12
  reducing contention 9-7
RECOVER option 6-16, 6-17
Recovering a damaged segment 9-35
Recovery procedures 1-19
Red Brick Decision Server
  component overview 1-5 to 1-7
  described 1-3
redbrick
  administrative user 1-15
  directory permissions 5-6
  user account 2-37, A-4
  user umask setting 5-6
Referenced tables
  defined 3-8
  outboard, outrigger 3-13
Referencing tables 3-8
Referential integrity
  defined 2-40
  delete behavior, defining 5-15 to 5-16
  delete operations 2-41
  load and insert operations 2-40
  ON DELETE clause 5-15
Relation scans 11-4, 11-26
REMOVE DAMAGED SEGMENTS option, ALTER DATABASE CLEAN VERSION LOG statement 6-19
RENAME clause, ALTER SEGMENT statement 9-30
Renaming segments 9-33
RENICE_COMMAND parameter
  rbw.config file entry B-25
  user priorities 8-12
REORG operations
  after ALTER SEGMENT 4-49
  effect of MAXROWS PER SEGMENT option 5-14
  effect of MAXSEGMENTS option 5-14
REORG_ANY task authorization 7-13
REPORT_INTERVAL parameter
  modifying 9-55, 9-58
  rbw.config file entry B-25
  usage 8-14
Resetting statistics 8-11
Resource privileges, with ODBC 7-5
RESOURCE system role
  described 2-38, 7-7
  granting 7-8, 7-18
  revoking 7-8, 7-21

Restore operations
during index creation 5-16
on versioned databases 6-24
Restricted delete 2-41
Restricted delete operations 5-15
RESTRICT_PREVIOUS parameter
rbw.config file entry B-26
usage 7-31
RESULT_BUFFER parameter
described 10-19
modifying 9-54, 9-57
rbw.config file entry B-17
RESULT_BUFFER_FULL_ACTION parameter
described 10-21
modifying 9-54, 9-57
rbw.config file entry B-17
Reuse by row 9-22
Reusing segments D-15
REVOKE authorization and role command 7-21
RISQL
described 1-4
RISQL calculate operator 10-54
RISQL Entry Tool
invoking A-8
user accounts 7-3
RISQL Entry Tool and RISQL Reporter
described 1-7
locales for 2-33
user accounts 7-4
Role-based security 7-10
Roles
direct and indirect membership 7-11
dropping 9-66
granting 7-8, 7-18 to 7-21
membership 7-18 to 7-21
revoking 7-8, 7-21
system 7-7
system table information 7-23
user-created 7-10 to 7-26
ROLE_MANAGEMENT task authorization 7-13
Row numbers
description 9-23
how database server uses 10-29
per block 10-31
RBW_ROWNUM pseudocolumn 9-31
RBW_SEGNAME pseudocolumn 9-23
rows per block 10-29, 10-35
specifying range 9-31, 9-32
unused 10-30, 10-31, 10-33, 10-34
VARCHAR fill factor effect 10-30
ROWCOUNT parameter
modifying 9-54, 9-57
rbw.config file entry B-21
Rows per block
calculation 10-29
CHECK TABLE output 10-35, 10-36
MAXROWS PER SEGMENT 10-34
number of row numbers 10-29
typical row size 10-29, 10-32
VARCHAR fill factor effect 10-31, 10-32, 10-33, 10-34
Rows, deleting
locking tables 9-10
referential integrity 2-41
ROWS_PER example 11-20
ROWS_PER_ equations 10-78, 11-18
ROWS_PER_FETCH_TASK parameter
described 11-4, 11-13
modifying 9-54, 9-57
rbw.config file entry B-16
syntax and usage 11-17
ROWS_PER_JOIN_TASK parameter
described 11-4, 11-13
modifying 9-54, 9-57
rbw.config file entry B-16
syntax and usage 11-17
ROWS_PER_SCAN_TASK parameter
described 11-4, 11-13
modifying 9-54, 9-57
rbw.config file entry B-16
syntax and usage 11-14

S

Sample scan operator 10-54
Sampling, query results 2-15
Schema events 8-23
Schema locking 9-7
Schemas. *See* Star schemas.
Scratch space, estimating requirements 4-36
Secondary dimension tables 3-13
Security
passwords 7-26
role-based 7-10
SEGMENT BY clause, ALTER SEGMENT statement 9-30
Segment parameters
entries in rbw.config B-22
Segmented STAR index picture D-12
Segmented storage
default location 10-22
described 2-7
implementation of 2-8
planning for use 4-45 to 4-49
See also Segments.
Segments
adding a new segment D-17
adding space to 9-26
altering 9-29 to 9-35
attaching 9-31
automatic locking 9-10
clearing 9-33
creation guidelines 4-48, 5-12
damaged 9-34
damaged, recovering 9-35 to 9-37
default
changing to named 4-46
defined 2-8
DEFAULT_SEGMENT_SIZE parameter 2-9
defined 2-7
detaching 9-31
distributing data among 2-10
dropping 9-67
intact 9-35
moving 9-31
named 2-7
named versus default 4-46

offline load, example D-20
online, offline 2-12, 9-32
synchronizing 9-32
optical, defined 9-38
partial availability 2-13
range, changing 9-32
renaming 9-33
reusing D-15, D-23
rolling off D-15
space available 9-18 to 9-20
time-cyclic data, example D-1 to D-23
verifying 9-34
SEGMENTS parameter
modifying 9-54, 9-57
rbw.config file entry B-22
usage 10-24
Selectivity, defined 4-10
SERIAL columns, changing start and step values for 9-45
Server monitor daemon. *See* rbw.servermon daemon.
SERVER parameter
modifying 9-53, 9-56
rbw.config file entry B-10
Server process 1-11
SERVER_NAME parameter
modifying 9-53, 9-56
rbw.config file entry B-11
Service, warehouse, overview of 1-9
Session
changing user priority 8-12
closing 8-11
SET commands, SQL
DEFAULT SEGMENT STORAGE PATH 10-22
IGNORE OPTICAL INDEXES 9-40
INDEX TEMPSPACE 10-12
OPTICAL AVAILABILITY 9-39
overview 2-24
PARTIAL AVAILABILITY 9-32, 10-26
QUERY MEMORY LIMIT 10-12
QUERY TEMPSPACE 10-12
REPORT_INTERVAL 8-15
RESULT BUFFER 10-19
RESULT BUFFER FULL ACTION 10-21
ROWS_PER_FETCH_TASK 11-20
ROWS_PER_JOIN_TASK 11-20
SEGMENT 10-24
specifying
STATS 10-5, 10-58
TMU COMMIT RECORD INTERVAL 6-5
TMU COMMIT TIME INTERVAL 6-5
TMU VERSIONING 6-17
TRANSACTION ISOLATION LEVEL 9-12
USE LATEST REVISION 6-22
VERSIONING 6-17
SET TRANSACTION ISOLATION LEVEL 9-12
Shared memory 1-14
SHMEM parameter
modifying 9-53
rbw.config file entry B-10
Simple merge operator 10-55
Size estimates
B-TREE indexes 4-26
example, database sizing 4-27
STAR indexes 4-26
system tables 4-32
tables and indexes 4-20
TARGET indexes 4-26
SKU number 3-16
Software dependencies 4
Sort 1-1 match operator 10-55
Sort component
defined 2-29
effect of changing 2-32
Sort phase of optimized index build 4-37
SQL statement, canceling 8-11
STAR indexes. *See* Indexes, STAR.
Star schemas 3-33
described 3-6
design considerations 3-18
examples 3-20, 3-33
performance 3-8
supported types 3-33
use of attributes 3-25
STARjoin
defined 2-6
parallel query performance 11-38
STARjoin operator 10-55
START ACCOUNTING clause, ALTER SYSTEM statement 8-41
Start and step values for SERIAL columns 9-45
START CLEANING clause, ALTER DATABASE statement 6-26
Starting, database server 9-60
Statistics
collection interval 8-14
platform effect on availability of 8-9
read and write 8-7
reporting 10-5
resetting 8-11
STOP ACCOUNTING clause, ALTER SYSTEM statement 8-41
STOP CLEANING clause, ALTER DATABASE statement 6-25
Stopping, database server 9-60
Storage devices, optical 9-37
Subquery in the FROM clause 10-85
Subquery operator 10-56
SuperScan technology 2-5, 11-6, 11-42
Support, customer 11
SWITCH ACCOUNTING FILE clause, ALTER SYSTEM statement 8-41
Synchronizing
segments 9-32
Synonyms
dropping 9-67
using with fact-to-fact joins 10-81
Syntax diagrams
conventions for 7
keywords in 9
variables in 9
System accounts, for new users 7-4
System catalog, contents C-2
System requirements 4

System roles
  described 7-7
  granting 7-8, 7-18
  overview 2-38
  revoking 7-8, 7-21
System tables
  described C-1 to C-25
  references to locale 2-31
  size estimates 4-32
System, database user account 5-9

T

Table Management Utility (TMU) 1-7
  Aroma examples A-24 to A-28
  enabling versioning 6-17
  loading with periodic commit 6-5
  TUNE parameters for B-16
Table scan operator 10-56
Tables
  access, limiting with views 5-20
  adding rows, effect on STAR index 4-49
  altering 9-40
  attaching segments 9-31
  browsing 3-24
  creating Aroma tables A-8
  creation guidelines
    foreign key order 5-14
    order 5-13
    referential integrity on delete 5-14
    setting MAXROWS PER SEGMENT 5-14
    setting MAXSEGMENTS 5-14
  dropping 9-67
  families, immediate and complete 2-41
  growth 4-48
  growth, monitoring 9-18
  in segments 5-12
  locking 9-6, 9-13
  multiple fact tables 3-10
  optical storage 9-39
  partial availability 2-13, 10-25
  segmenting column 9-32
  size estimates 4-21
  sizing 4-21
  system tables, size estimates 4-32
  transient 4-48
  when to lock 9-10
TARGET indexes. *See* Indexes, TARGET.
TARGET scan operator 10-57
TARGETjoin
  defined 2-6
  operator 10-56
  planning for 4-12
  query processing 10-61 to 10-77
  rules 10-62
  when to use 10-64
Task authorizations
  ACCESS_SYSINFO 8-4
  ALTER_SYSTEM 8-4
  defined 7-12
  for managing database activity 8-4
  granting 7-16
Tasks for parallel queries, allocating 11-23 to 11-29
Technical support 11
Temporary macros
  deleting 9-66
  scope 5-23
Temporary space 4-35, 4-44, 10-7 to 10-18
Temporary tables 4-21
TEMPORARY_DATA_SEGMENT parameter
  modifying 9-54, 9-57
  rbw.config file entries B-22
TEMPORARY_INDEX_SEGMENT parameter
  modifying 9-54, 9-57
  rbw.config file entries B-22
TEMP_RESOURCE task authorization 7-13
Territory
  defined 2-28
  effect of changing 2-32, 2-36
Thread, defined 1-9
Time-cyclic data, example D-1 to D-23
Tip icons 10
TMU. *See* Table Management Utility (TMU).
TMU_BUFFERS parameter
  modifying 9-55, 9-58
  rbw.config file entry B-16
TMU_COMMIT_RECORD_INTERVAL parameter
  modifying 9-55, 9-58
  rbw.config entry B-14
  using for periodic commit 6-5
TMU_COMMIT_TIME_INTERVAL parameter
  modifying 9-55, 9-58
  rbw.config entry B-14
  using for periodic commit 6-5
TMU_CONVERSION_TASKS parameter
  modifying 9-55, 9-58
  rbw.config file entry B-19
TMU_INDEX_TASKS parameter
  modifying 9-55, 9-58
  rbw.config entry B-20
TMU_INPUT_TAKS parameter
  modifying 9-58
TMU_INPUT_TASKS parameter
  modifying 9-55
  rbw.config entry B-20
TMU_MAX_TASKS parameter
  modifying 9-55, 9-58
  rbw.config entry B-20
TMU_MMAP_LIMIT parameter
  modifying 9-55, 9-58
  rbw.config entry B-20
TMU_OPTIMIZE parameter
  modifying 9-55, 9-58
  rbw.config file entry B-18
TMU_ROW_MESSAGES parameter
  modifying 9-55, 9-58
  rbw.config entry B-19
TMU_SERIAL_MODE parameter
  modifying 9-55, 9-58
  rbw.config entry B-19
TMU_VERSIONING parameter
  enabling versioning 6-17
  modifying 9-55, 9-58
  rbw.config file entry B-14
  RECOVER option 6-17
TOTALFREE column, RBW_SEGMENTS table 9-22

TOTALQUERYPROCS parameter
definition 11-4
described 11-5 to 11-27
modifying 9-53, 9-56
rbw.config file entry B-16
syntax and usage 11-10
Transaction-processing databases 3-4
Transactions
defined 2-39
single statement 2-39
TRANSACTION_ISOLATION_LEVEL parameter
modifying 9-54, 9-57
rbw.config file entry B-13
Transient tables 4-48
Translated text 2-32
Trickle feed applications 6-5
Troubleshooting, general 13
TUNE parameters
entries in rbw.config B-16
FORCE_AGGREGATION_TASKS 11-31
FORCE_FETCH_TASKS 11-24
FORCE_JOIN_TASKS 11-24
FORCE_SCAN_TASKS 11-24
INDEX_TEMPSPACE_DIRECTORY 10-12
INDEX_TEMPSPACE_MAXSPILLSIZE 10-12
INDEX_TEMPSPACE_THRESHOLD 10-12
OPTICAL_AVAILABILITY 9-39
PARTITIONED_PARALLEL_AGGREGATION 11-31
QUERY_MEMORY_LIMIT 10-12
QUERY_TEMPSPACE_DIRECTORY 10-12
QUERY_TEMPSPACE_MAXSPILLSIZE 10-12
summary of B-28
Tuning
database 1-19
parallel query parameters 11-4

U

UNFREEZE QUERY REVISION clause, ALTER DATABASE statement 6-22
UNIFIED_LOGON parameter
modifying 9-56
rbw.config file entry B-10
UNIFORM_PROBABILITY_FOR_ADVISOR parameter
modifying 9-54, 9-57
rbw.config file entry B-13
UNION versus OR operation 10-84
Unloading data
frozen versions 6-22
overview 1-18
UPC level 3-23
Update operator 10-57
UPGRADE_DATABASE task authorization 7-13
Usage events 8-23
USAGE ROUTINE event 8-23, 8-29
USED column, RBW_STORAGE table 9-21
User access, providing 1-17
User accounts
adding new users 7-4 to 7-6
expired 7-29
locked 7-36
RISQL Entry Tool 7-3
User priority, changing 8-12
User-created roles
creating 7-15
direct and indirect membership 7-11
dropping 9-66
granting 7-18 to 7-21
managing 7-10 to 7-26
revoking 7-21
Users
adding to a database 7-5
changing passwords 7-6
types of 3
User, redbrick 1-15, 2-15, 2-37
USER, SQL variable 5-21
USER_MANAGEMENT task authorization 7-13
USE_INVALID_PRECOMPUTED_VIEWS parameter
modifying 9-54, 9-57
rbw.config file entries B-13
Utility programs
copy management 8-4
dbsize 4-21
findserver 9-61
rb_cm 8-4
See also dbsize utility, dbcreate utility, rb_creator utility, and rb_deleter utility.

V

Vacuum cleaner daemon
controlling 6-25
description 1-13
manually starting 6-26
manually stopping 6-25
VARCHAR columns
changing from CHAR 9-43
fill factor effects 10-29
modifying fill factor 10-37
monitoring fill factor 10-35
setting fill factor for 10-29
Variables in syntax diagrams 9
VERIFY clause
ALTER SEGMENT statement 9-37
VERIFY clause, ALTER SEGMENT statement 9-30, 9-34
damaged segments 9-35
Version log
adding space to 6-20
allocating space 6-18
blocks of data in 6-11
cleaning 6-19
costs 6-7
creating 6-18
deleted blocks in 6-13
dropping 6-20
enabling 6-16
high recoverability for 6-16
monitoring 6-23
monitoring databases 6-23
PSUs for 6-11

sizing 6-18
structure of 6-11
tuning for performance 6-18
understanding 6-10
using as a write-ahead log 6-16
verifying if empty 6-19
Version of database server software 9-64
Versioned databases
Aroma example 6-26
described 2-39
Versioned transactions, defined 2-39
Versioning
cleaning dimension tables 6-7
controlling the vacuum cleaner 6-25
cost of 6-7
database availability 6-4
deleting blocks 6-13
determining if needed 6-4
enabling 6-16
frozen versions, controlling 6-21
frozen versions, understanding 6-13
high recoverability and frozen versions 6-22
high recoverability option 6-16
loading data with 6-8
options for 6-16
periodic commit option 6-5
with Vista 6-6
VERSIONING parameter
enabling versioning 6-17
modifying 9-54, 9-57
rbw.config file entry B-13
Views
creation guidelines 5-19
described 3-17
dropping 9-68
precomputed 2-14
Virtual table scan operator 10-57
Vista
Advisor logging commands 8-29
aggregate maintenance 2-14
described 2-14
query sampling 2-15
versioning with 6-6
Visual Basic, database access with 7-5

W

WAIT / NO WAIT on locks 9-11
Warehouse. *See* Database server.
Warehouse processes
CTRL-C coordination thread 1-14
daemon process 1-11
listener thread 1-13
log daemon process 1-12
overview 1-9 to 1-14
process checker daemon 1-12
server process 1-11
vacuum cleaner daemon
Warehouse service on Windows
defined 1-11
monitoring 9-62
starting 9-63
Warning icons 10
Weak selectivity
defined 4-10
improving performance for queries with 5-19
Workload accounting 8-34
WRITE locking option 9-6
Write locks 9-8
Write statistics 8-9

Symbols

.rbretrc file 1-17
.rbwerr file 2-23
.rbwrc file 1-17, 2-22